AFRL-IF-RS-TR-1999-61
Final Technical Report
April 1999

FORMAL METHODS FOR INTEGRATING KNOWLEDGE BASES

Kestrel Institute

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. C322

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1999-61 has been reviewed and is approved for publication.

APPROVED: Raymond A. Liuzzi

RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR: Northrup Fowler

NORTHRUP FOWLER, III, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

FORMAL METHODS FOR INTEGRATING KNOWLEDGE BASES

David Espinosa

Contractor: Kestrel Institute
Contract Number: F30602-95-C-0122
Effective Date of Contract: 17 April 1995
Contract Expiration Date: 30 December 1998
Short Title of Work: Formal Methods for Integrating Knowledge Bases

Period of Work Covered: Apr 95 - Dec 98

Principal Investigator: David Espinosa
Phone: (650) 493-6871
AFRL Project Engineer: Raymond A. Liuzzi
Phone: (315) 330-7796

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Raymond A. Liuzzi, AFRL/IFTB, 525 Brooks Road, Rome, NY 13441-4505.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1999	Final Apr 95 - Dec 98

4. TITLE AND SUBTITLE

FORMAL METHODS FOR INTEGRATING KNOWLEDGE BASES

5. FUNDING NUMBERS

C - F30602-95-C-0122
PE - 62301E
PR - C322
TA - 00
WU - 01

6. AUTHOR(S)

David Espinosa

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Kestrel Institute
3260 Hillview Avenue
Palo Alto CA 94304

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

Air Force Research Laboratory/IFTB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-1999-61

11. SUPPLEMENTARY NOTES

Air Force Research Laboratory Project Engineer: Raymond A. Liuzzi/IFTB/(315) 330-3577

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The object of this effort is to demonstrate the effectiveness of formal specification and refinement techniques for constructing realistic mediators. Such techniques will enable the rapid and reliable construction of mediators in fast-breading situations. This report describes the problem of mediation and an approach to solving this problem, that includes a formal specification and refinement process for mediator generation. SPECWARE is a formal software development tool that is extended with a mediator generation capability. This effort describes the process of translating specifications to code in Lisp or C++. The facility described is used both for describing wrappers and generating mediation code. Formal wrappers were built in SPECWARE. The theoretical notion of "patching" provides a systematic way of handling multiple representations of the same concept which is a basic problem in mediation. Patching, and its implementation in SPECWARE, are the major contributions of this project to mediation technology. This report also describes three demonstrations and summarizes the results, and outlines future work.

14. SUBJECT TERMS

Data Base, Knowledge Base, Software, Artificial Intelligence, Computers

15. NUMBER OF PAGES

80

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

1 Introduction **1**
1.1 Outline 1

2 Background and Project Overview **1**
2.1 The Mediation Problem 1
2.2 Objective 2
2.3 Focus 2
2.4 Approach 2

3 Specware Overview **4**
3.1 Specware, a Formal Software Development Environment 4
3.2 Current Capabilities of Specware 5
3.3 Composition with Overlaps 5

4 Code Generation **10**
4.1 Entailment Systems and their Morphisms 10
4.2 Localizing Logic Morphisms 12
4.3 C++ Code Generation 13

5 Formal Wrappers **15**
5.1 Wrapping the GNIS database 15
5.2 Wrapping the DEM database 24

6 Patching Multiple Representations **25**
6.1 An Analogy between Information Integration and Manifolds 25
6.2 An Example of Patching: Multiple Representations of Angles 25
6.3 Patching as a Composition Operator 27

7 Demonstrations **34**
7.1 Scheduling demonstration 35
7.2 SQL demonstration 38
7.3 GIS demonstration 48

8 Results and Future Plans **57**

References **59**

1 Introduction

This document constitutes the final progress report on "Formal Methods for Integrating Knowledge Bases", Contract No. F30602-95-C-0122 to Rome Laboratory. Technical work performed under this contract from 17 April 1995 to 31 December 1998 is summarized in this report.

1.1 Outline

Section 2 provides some background on the problem of mediation and describes our approach to solving this problem, a formal specification and refinement process for mediator generation. Section 3 describes SPECWARE, a formal software development tool being built at Kestrel. In this project, SPECWARE is being extended with a mediator generation capability. Section 4 describes the process of translating specifications to code in Lisp or C++. This facility is used both for describing wrappers and generating mediation code. Section 5 discusses how formal wrappers are built in SPECWARE. Section 6 describes the theoretical notion of "patching", which provides a systematic way of handling multiple representations of the same concept, a basic problem in mediation. Patching, and its implementation in SPECWARE, are the major contributions of this project to mediation technology. Section 7 describes three demonstrations of the work done under this project. Section 8 summarizes the results so far and outlines future work.

2 Background and Project Overview

2.1 The Mediation Problem

The proliferation of computers and the phenomenal advances in interconnectivity via high speed networks have resulted in easy access to a large number of information sources. Consequently, modern applications, both military (e.g., battlefield management) and commercial (e.g., airline scheduling), depend on pulling together information from several different sources. The *heterogeneity* of the information sources, and the applications using them, is a significant hurdle to the effective use of the information. Heterogeneity arises in several ways: different computing platforms, different representation and programming languages, and different semantic assumptions.

Mediation is the problem of providing a coherent information conduit between a collection of heterogeneous sources and applications. This entails translation between different representations as well as the reconciliation of the same information represented differently by several sources. Such translation and reconciliation can range from syntactic (e.g., conversion between different data formats) to semantic (e.g., relating different kinds of 'altitude').

A mediator is an (extra) application which mediates all transactions between the information sources and applications to which it is connected. It presents a single, semantically coherent view of the information sources to the applications; in other words, it hides the heterogeneity of the information sources from the applications by performing all the necessary translation and reconciliation.

Project Overview

2.2 Objective

The objective of our project is to demonstrate the effectiveness of formal specification and refinement techniques for constructing realistic mediators. Such techniques will enable the rapid and reliable construction of mediators in fast-breaking situations.

2.3 Focus

Within the larger problem of mediation, our project focusses on the aspect of *semantic interoperation*, i.e., relating information from heterogeneous sources at a semantic level. We further focus on articulating paradigmatic methods of semantic interoperation, so as to enable the generation of mediators.

2.4 Approach

Our approach is to apply a formal specification and refinement process to the development of mediators. To this end, information sources are first wrapped in formal interface specifications; the application(s) are also similarly wrapped. We assume the availability of libraries of specifications (ontologies) suitable for this purpose. Next, the interface specifications are composed, with conversion functions inserted for shared parts; the composition represents the global interface specification for the information sources. Finally, the operations in the application interface are realized in terms of (or, refined into) the global interface specification. Figure 1 renders this process pictorially, and highlights features of this approach.

The implementation task of this project is being carried out using SPECWARE, a tool that supports the modular construction of formal specifications and their refinement to code (in Lisp or C++).

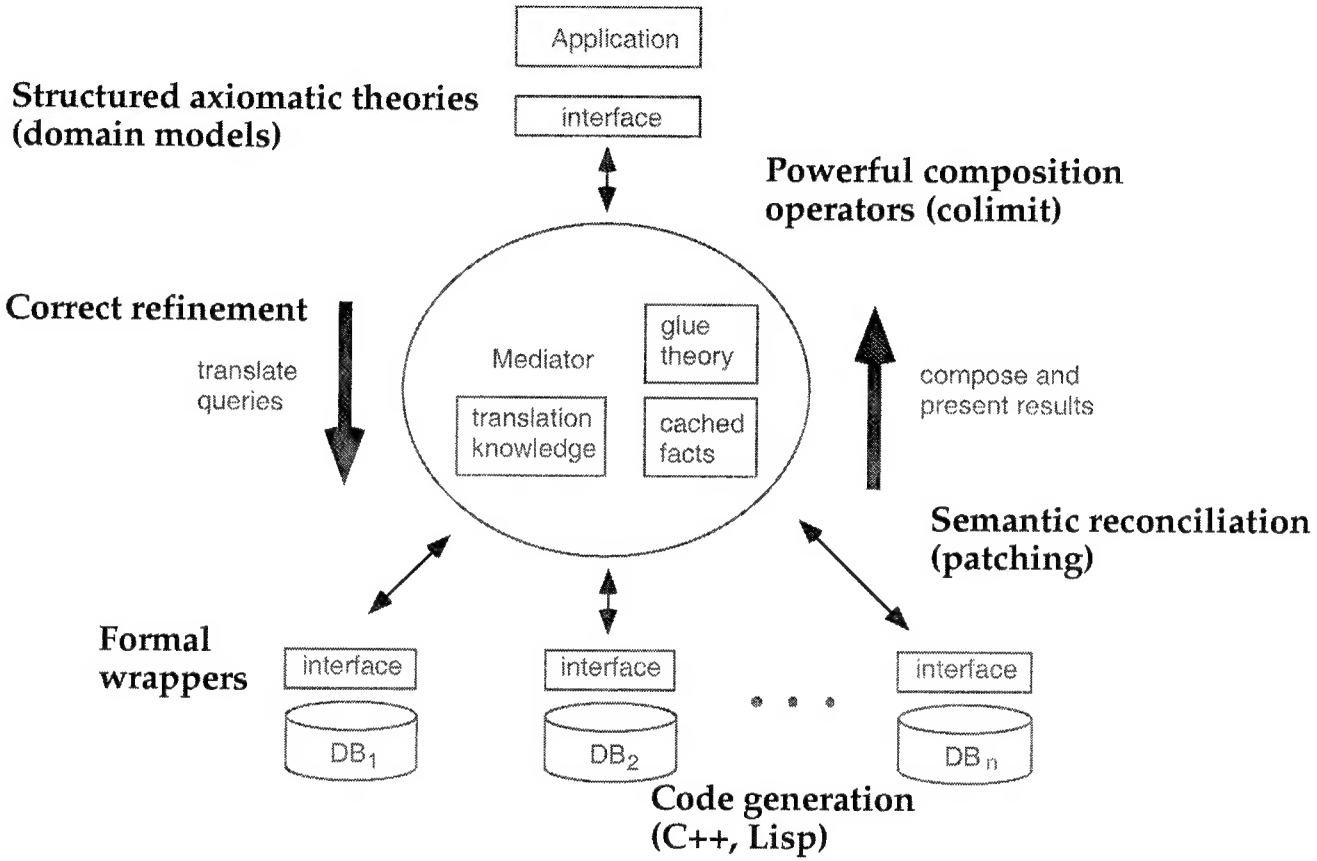

Figure 1: Features of the Kestrel approach to mediation

3 Specware Overview

3.1 Specware, a Formal Software Development Environment

SPECWARE [Srinivas and Jüllig 95] is a tool that supports the modular construction of formal specifications (in higher order logic) and the stepwise and componentwise refinement of such specifications into executable code (in Lisp and C++). Software development in SPECWARE is characterized by two tenets:

Description: We always deal with descriptions, i.e., a collection of properties, of the artifact that we ultimately wish to build. These descriptions are progressively refined by adding more properties, until we can exhibit a model or witness (usually a program) which satisfies these properties. Descriptions in SPECWARE are written in one of several logics.

Composition: We handle complexity and scale by providing composition operators which allow bigger descriptions to be put together from smaller ones. The colimit operation from category theory is pervasively used for composing structures of various kinds in SPECWARE. Besides composition operators, one needs bookkeeping facilities and information presentation at various abstraction levels. SPECWARE uses category theory for bookkeeping and abstraction.

SPECWARE maintains a design history which indicates how the final code is derived from the original specification. In this manner, the advantage of a declarative, knowledge-based approach is combined with the efficiency of optimized code.

SPECWARE is a shift from formality in-the-small to formality in-the-large. This shift has necessitated a new conceptual basis in category theory, topology, and sheaf theory, abstract mathematical theories that were originally invented for dealing with complex structures. On top of this mathematical kernel, SPECWARE uses algebraic specification and general logics, formalisms which have resulted from decades of research in formal specification. These formalisms are based on category theory and provide abstract composition operators which are independent of the specification language. The language of category theory results in a highly parameterized, robust, and extensible architecture that can scale to system development.

History. Kestrel has been pursuing a knowledge-based approach to software development for over a decade using KIDS, an algorithm design system [Smith 90], DTRE, a data type refinement system [Blaine and Goldberg 91], and REACTO, a state-machine design system [Gilham et al. 89]. The common thread in these tools is the explicit representation of knowledge: foundation knowledge (data types, arithmetic, etc.), domain-specific knowledge (transportation resource models, scheduling

constraints, etc.), and problem-solving knowledge (divide-and-conquer, global search, incremental computation, etc.) SPECWARE is an attempt to integrate the capabilities of these tools on the common conceptual foundation of structured theories, and moreover, provide much more functionality in a scalable and extensible way.

3.2 Current Capabilities of Specware

Specifications in SPECWARE are written in a variant of higher order logic called SLANG [Srinivas and Jüllig 95, Lambek and Scott 86]. Specifications can be built modularly via specification-building operations such as import, translate and colimit. One specification can be refined into another (the latter being less abstract or more concrete) via an interpretation [Lambek and Scott 86, Turski and Maibaum 87]. An interpretation formally indicates how the types and operations of one specification are realized in terms of the types and operations of another specification.

Interpretations can be cascaded, thus resulting in stepwise refinement. Moreover, interpretations interact gracefully with the specification-building operations: a specification built from parts can be refined by refining its parts in a compatible way. There is thus a two-dimensional space of specifications related by the "part-of" relation in one dimension and the refinement relation in the other dimension.

A sufficiently refined specification can be transformed into executable code in programming languages such as Lisp and C++. This process is represented in SPECWARE as refinement into a different logic, i.e., programs are specifications too! Again, such inter-logic refinements can be composed.

Figure 2 shows the graphical interface of SPECWARE and highlights various capabilities. A resolution prover provides inference services. A library of foundation theories, e.g., containers, algebraic structures (monoids, groups, partial orders, etc.), numbers, etc., is preloaded into SPECWARE. Domain-specific libraries for transportation scheduling (e.g., tasks, resources) and problem-solving libraries (e.g., divide-and-conquer, global search) are also being added.

3.3 Composition with Overlaps

Interesting interconnections of parts have overlaps. In SPECWARE, interconnections of components are represented by diagrams (a formal notion in category theory) of objects related by arrows which indicate the overlaps. The colimit operation produces a single object from the diagram by "gluing" the parts together along the indicated overlaps. Conversely, the diagram may be construed as a "covering" (a formal notion in topology) of the colimit object by parts. This kind of composition is abstractly illustrated in Figure 3.

Composition using diagrams and colimits is more general than the commonly used mechanisms of import and inheritance. Morphisms provide precise control over how

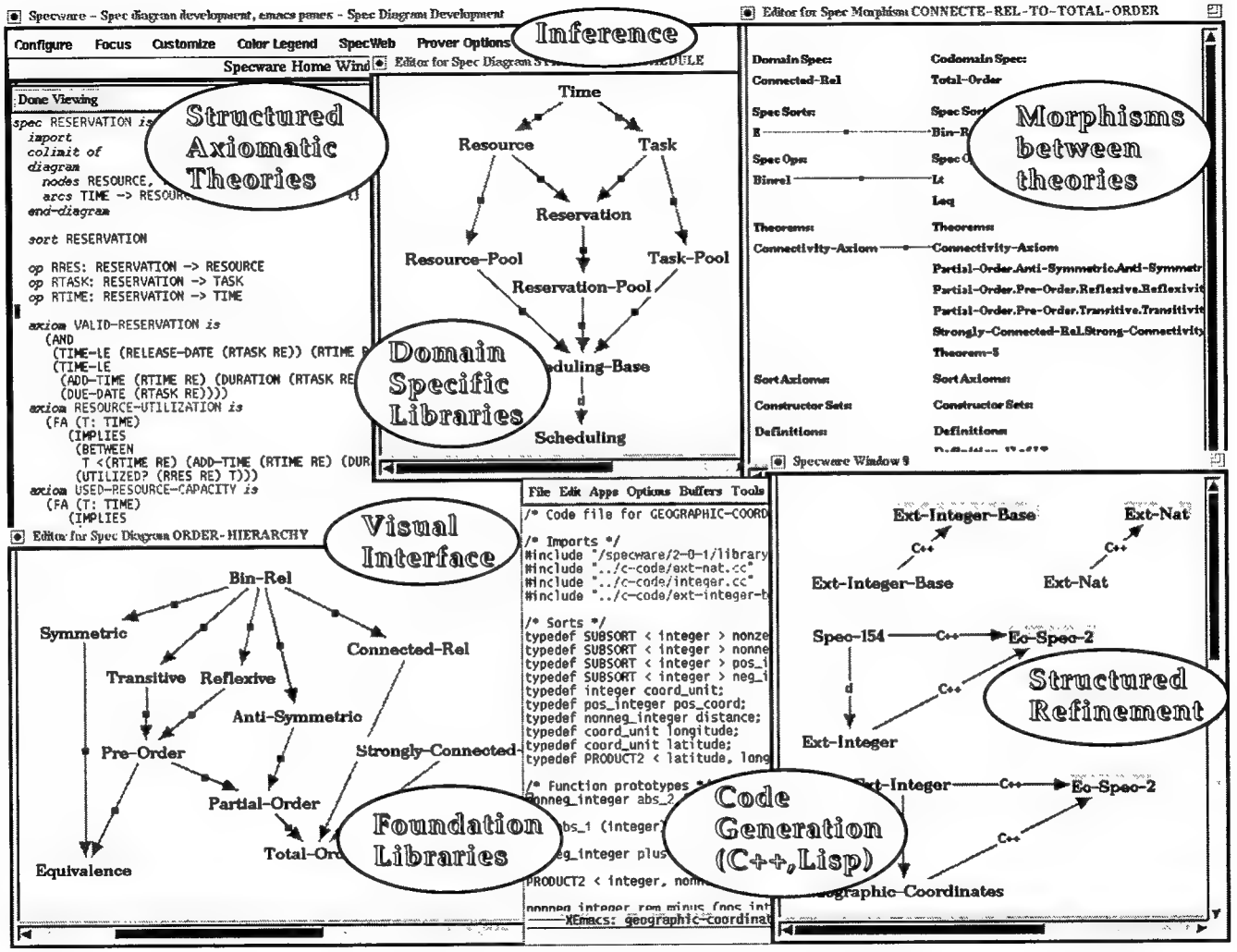

Figure 2: Capabilities of Specware

the parts are related and indicate which parts are to identified and which are to be copied. Import and inheritance, on the other hand, rely on implicit rules to determine sharing and ancestry, a reliance which is not conducive to scaling.

Example (specification composition). Figure 4 shows a simple example of a specification composed from smaller specifications via a colimit operation (following the abstract pattern in Figure 3). The specifications describe 1-degree digital elevation models (`DEM1`), which are regularly spaced grids of elevations on the earth's surface. `DEM1-COLUMN` is the specification for a column of elevations (i.e., in the North-South direction). `DEM1-RECTANGLE` specifies grids of elevations, as arrays of columns. `DEM1-WRAPPER` is the specification of functions which retrieve single columns from a file containing elevation data. As a step towards providing a higher-level interface for elevation models, `DEM1-RECTANGLE` and `DEM1-WRAPPER` are composed, with `DEM1-COLUMN` being the part shared between them.

3.3.1 Structured Refinement

SPECWARE provides composition operators not only for specifications, but also for refinements. This is done by lifting the structure of specifications to refinements: a composed object can be transformed or refined using transformations of the parts, provided these are "compatible" (a formal notion in sheaf theory), i.e., the sub-transformations agree where the parts overlap. Refinement with overlaps is abstractly illustrated in Figure 5.

Structured refinement is realized in SPECWARE via the notion of diagram refinement, which is a diagram of refinements and refinement morphisms connecting two specification diagrams. A diagram refinement can be parallely composed to obtain a refinement from the colimit of the source diagram to the colimit of the target diagram. Diagram refinements are thus a structuring mechanism for refinements. This structuring mechanism is mostly independent of the particular notion of "refinement"; hence, it applies not only to refinements between specifications, but also to refinements from specifications to code.

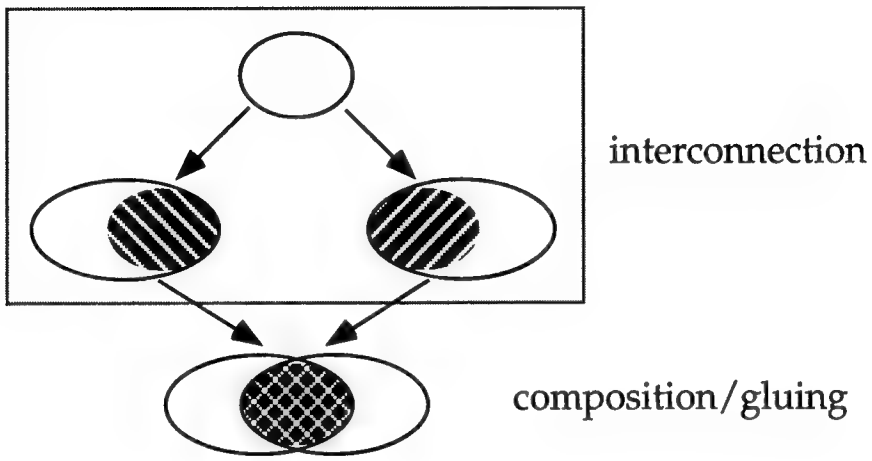

Figure 3: Composition with Overlaps

```
spec DEM1-INTERFACE-base is
  colimit of diagram
    nodes DEM1-COLUMN, DEM1-RECTANGLE, DEM1-WRAPPER
    arcs  DEM1-COLUMN -> DEM1-RECTANGLE : {},
          DEM1-COLUMN -> DEM1-WRAPPER   : {}
  end-diagram
```

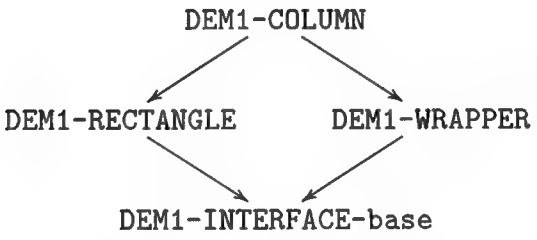

Figure 4: An example of specification composition

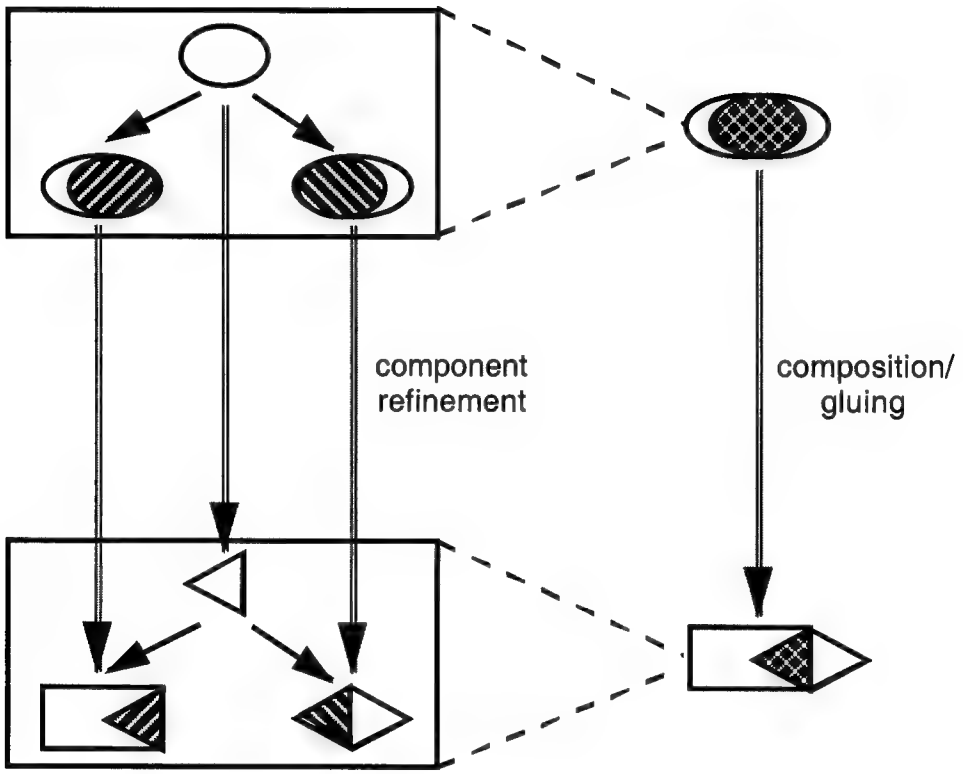

Figure 5: Refinement with Overlaps

4 Code Generation

Code generation is the process of translating a specification into a code module in a programming language while preserving semantics. In the mediation context, code generation makes the integrated interface specification executable.

Code generation in SPECWARE is represented as a logic morphism from the logic of the specification language SLANG to the logic of a target language such as Lisp or C++. A logic morphism translates the syntax of one logic into the syntax of another, while preserving the semantics [Meseguer 89].

In general, there may be several logic morphisms between any two logics or languages. On a given language fragment, the different logic morphisms typically yield translations with different performance characteristics. It is desirable to translate one part of a language via one logic morphism and another part via another morphism. Sheaves and diagram refinements provide a systematic way of combining logic morphisms. In other words, code generation is just another kind of refinement; so, the generic machinery for structured refinement is applicable.

We describe below the theoretical underpinnings of code generation in SPECWARE because logic morphisms are the basis for establishing the correctness of a wrapper. Once all the information sources relevant to a problem are wrapped, then we can work entirely within the logic of SPECWARE and reason about properties of interest. Consistency of the integrated interface specification will then imply the correctness of the generated mediator.

4.1 Entailment Systems and their Morphisms

A logic consists of an entailment system (syntax and provability) and an institution (semantics) suitably related. We will not need institutions for code generation in SPECWARE. The definitions below are from [Meseguer 89].

An entailment system consists of signatures, sentences and an entailment (or deduction) relation. Signatures typically consist of types and operations, which provide the vocabulary for describing a domain of interest. Each logic also defines what its well-formed sentences (or formulas) are; these are used to describe the properties of the types and operations in a domain. The deduction relation allows us to reason about these properties.

DEFINITION 4.1: *Entailment System.* An entailment system is a 3-tuple $\langle \mathbf{Sig}, sen, \vdash \rangle$ consisting of

1. a category **Sig** of signatures and signature morphisms,
2. a functor $sen: \mathbf{Sig} \to \mathbf{Set}$ (where **Set** is the category of sets and functions) which assigns to each signature Σ the set of Σ-sentences, and to each signature morphism $\sigma: \Sigma \to \Sigma'$, the function which translates Σ-sentences to Σ'-sentences (this function will also be denoted by σ), and
3. a function $\vdash$ which associates to each signature Σ a binary relation between sets of sentences and sentences $\vdash_\Sigma \subseteq \mathcal{P}(sen(\Sigma)) \times sen(\Sigma)$, called Σ-entailment,

such that the following properties are satisfied:

1. *reflexivity:* for any $\varphi \in sen(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$;
2. *monotonicity:* if $\Gamma \vdash_\Sigma \varphi$ and $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash_\Sigma \varphi$
3. *transitivity:* if $\Gamma \vdash_\Sigma \varphi_i$, for $i \in I$, and $\Gamma \cup \{\varphi_i \mid i \in I\} \vdash_\Sigma \psi$, then $\Gamma \vdash_\Sigma \psi$;
4. $\vdash$*-translation:* if $\Gamma \vdash_\Sigma \varphi$, then for any signature morphism $\sigma: \Sigma \to \Sigma'$, $\sigma(\Gamma) \vdash_{\Sigma'} \sigma(\varphi)$.

□

To map one entailment system into another, we map the syntax while preserving entailment. A simple way to map syntax is to map signatures to signatures, and sentences over a signature to sentences over the translated signature. If the former is a functor, the latter becomes a natural transformation.

DEFINITION 4.2: *Entailment system morphism (plain version).* A morphism between entailment systems $\langle \Phi, \alpha \rangle: \langle \mathbf{Sig}, sen, \vdash \rangle \to \langle \mathbf{Sig}', sen', \vdash' \rangle$ is a pair consisting of a functor $\Phi: \mathbf{Sig} \to \mathbf{Sig}'$ which maps signatures to signatures and a natural transformation $\alpha: sen \dot{\to} sen' \circ \Phi$ which maps sentences to sentences such that entailment is preserved:

$$\Gamma \vdash_\Sigma \varphi \Rightarrow \alpha_\Sigma(\Gamma) \vdash'_{\Phi(\Sigma)} \alpha_\Sigma(\varphi).$$

□

Morphisms which map signatures to signatures are not flexible enough, especially for code generation. In general, it may be necessary to map built-in elements of one logic into defined elements of another, and vice versa. This can be realized by mapping signatures to specifications, and vice versa, or, in general, specifications to

specifications. However, morphisms which map specifications to specifications are too unconstrained, so we impose the restriction that there be an underlying map of signatures. This gives the right amount of flexibility in mapping sentences. We will omit the general definition of entailment system morphism (see [Meseguer 89] for details).

4.2 Localizing Logic Morphisms

Logic morphisms (and entailment system morphisms) map entire categories of specifications at once. In the SPECWARE context, we want to be able to translate different specifications via different morphisms. In other words, we want to build a logic morphism from pieces of other logic morphisms. This is achieved in SPECWARE by defining a new kind of arrow, *inter-logic specification morphism*, which localizes the action of a logic morphism to a single specification. These arrows then participate in the normal SPECWARE modularization mechanisms of sequential and parallel composition.

4.2.1 Inter-Logic Specification Morphism

An inter-logic specification morphism connects specifications in different logics. It is similar to a specification morphism in that it maps the source signature to the target signature such that source axioms translate to target theorems. The difference is that the structure of the source and target signatures may be different (e.g., Lisp specifications do not have sorts), and built-in entities in the source may map to defined entities in the target (e.g., the SLANG built-in "implies" is not part of Lisp). Similarly the structure of source and target axioms (sentences) may be different (e.g., SLANG operations always take a single argument and return a single result, whereas Lisp and C++ operations are n-ary and generally return only one result). Polymorphic operations in one logic may translate to families in another logic, and vice versa. For example, SLANG equality (which is polymorphic) translates to a family of equalities in Lisp, while families of List operations translate to the "polymorphic" List operations of Lisp.

4.2.2 Inter-Logic Interpretation

An inter-logic interpretation is similar to an interpretation except that the source and target specifications are in different logics. An inter-logic interpretation is a pair consisting of an inter-logic specification morphism and a definitional extension in the target specification category, such that the codomains of the two arrows match.

4.3 C++ Code Generation

To generate C++ code, we must view C++ as a logic (described below) and specify how SLANG concepts translate to C++ concepts. Sorts in SLANG map to types or classes in C++. Sort constructors, e.g., subsort, quotient, etc., map to templates. SLANG operations map to C++ operations of corresponding type. Definitions maps to definitions. The translation of a structured specification is obtained from the translations of its components via the refinement composition operators of SPECWARE.

4.3.1 C++ Specifications

C++ specifications consist of types, constants, operations and definitions. In addition, a set of files containing C++ code may be associated with a C++ specification. The types, constants, operations and definitions in these files are considered to be part of the C++ specification although not explicitly represented.

4.3.2 C++ Specification Morphisms

C++ specification morphisms are similar to SLANG specification morphisms in that they map types to types, constants to constants, operations to operations and definitions to definitions. We distinguish three kinds of morphisms:

import morphisms, which include one specification into another,

parameter morphisms, which exhibit the parameter to a parameterized specification, and

instantiation morphisms, which bind a parameter specification to an actual specification.

All morphisms, except instantiation morphisms, are injective.

4.3.3 Implicit Sharing

Import morphisms are construed as inclusions in C++. Thus, when C++ specifications are combined via colimits, the common imports are automatically identified.

4.3.4 Names in C++ Specifications

The type and constant names in a C++ specification are required to be unique. Operation names may be overloaded (in the C++ sense). When generating C++ code, the system tries to preserve type and operation names used in the source (SLANG) specification, unless there is a name clash, in which case the system chooses a unique name.

4.3.5 *Slang to C++ Morphisms and Interpretations*

A SLANG-to-C++ (inter-logic) morphism consists of two maps: one which maps SLANG sorts to C++ types, and one which maps SLANG operations to C++ operations.

A SLANG-to-C++ (inter-logic) interpretation is a pair consisting of a SLANG-to-C++ morphism and a C++ import morphism.

4.3.6 *Adding a New Basic Translation*

The system provides syntax for adding primitive SLANG-to-C++ interpretations. The user has to provide a target specification which imports `SLANG-BASE` (the translation of SLANG built-ins), and two maps, one which translates sorts and one which translates operations.

This is how external information sources, together with code which accesses them, are made visible to SPECWARE. Details are provided in Section 5.

5 Formal Wrappers

In SPECWARE, a wrapper is simply an interpretation that implements a high-level interface in terms of a low-level interface. The wrapper takes an application described by the low-level interface and re-presents it via the high-level interface. Wrappers are typically used to:

- add functionality,
- simplify semantics,
- translate between representations, and
- change languages.

The full SPECWARE functionality for constructing interpretations is available for building wrappers. That is, interpretations may be constructed via sequential and parallel composition of other interpretations, as described in Section 3.

In this section, we discuss the wrapping of two databases available from the US Geological Survey (USGS):

- Geographic Name Information System (GNIS), a tabular database describing named geological features throughout the US, and
- Digital Elevation Model (DEM), a custom format database providing elevation data in one-degree squares, also throughout the US.

5.1 Wrapping the GNIS database

The GNIS database can be obtained in two formats, a concise format containing basic information about larger geographic features all over the US, and a detailed format containing much more data, available per state. In the GIS demo, we work with concise data for California only (250 kb), extracted from the US concise file (5 mb). For comparison, the detailed California data occupies 18 mb.

Figure 6 shows an excerpt from the GNIS concise database. The USGS data arrives in ASCII format with fixed size records, redundantly delimited by carriage returns. The fields in order are name, type, county, location, and elevation. For variety, the figure shows many of the different types of places that may occur; in actuality, most of the file describes populated places (type `ppl`).

The GNIS wrapper is extremely simple. It changes language (from C++ to SLANG) and representation (from natural numbers to representation independent global locations, as discussed in Section 6), but it does not add functionality or simplify the

```
Acton                          ppl       Los Angeles     342812N1181145W 2688
Agassiz, Mount                 summit    Fresno          370642N1183148W 13891
Agua Caliente Reservation      reserve   Riverside       334600N1163400W
Agua Hedionda                  bay       San Diego       330833N1171936W
Alameda County                 civil     Alameda         373600N1215300W
Alamo River                    stream    Imperial        331244N1153715W
Alcatraz Island                island    San Francisco   374936N1222520W
All American Canal             canal     Imperial        324219N1150328W
Almanor, Lake                  reservoir Plumas          401023N1210515W 4500
Amargosa Range                 range     Inyo            363000N1164200W
Anacapa Passage                channel   Ventura         340058N1192747W
Angeles National Forest        forest    Los Angeles     341800N1180800W
Ano Nuevo, Point               cape      San Mateo       370647N1221945W
Antelope Valley                valley    Los Angeles     344500N1181500W
Arrowhead, Lake                lake      San Bernardino  341552N1171104W 5114
Badwater Basin                 basin     Inyo            361500N1164930W
```

Figure 6: Sample GNIS data

semantics, since all needed functionality is already available in a simple form in the low-level interface.

The GNIS wrapper is formed by the parallel composition of three parts:

- an interface for GNIS records,
- an interface for searching GNIS files, and
- extra operations needed from the ontology library.

We will discuss these in turn.

5.1.1 GNIS records

Since the GNIS record interface is just a simple record type, it is actually generated mechanically from a description of the record fields, shown in Figure 7. The numbers in the figure are the locations and lengths of the fields within the ASCII record. It is basically unimportant that the interface is mechanically generated; we would have written exactly the same specifications by hand. We describe each of the components of the interface, starting from the actual C++ code.

GNIS records are represented by the C++ code shown in Figure 8. The figure shows only the representation type, omitting the code for the actual operations. This

code is connected to the C++ specification `GNIS`, shown in Figure 9, via a reference to the code file `gnis.cc`.

The next level of wrapping is the SLANG specification `GNIS`, shown in Figure 10. This specification is essentially identical to the C++ `GNIS` specification and serves only to change languages from C++ to SLANG. Although the SLANG specification does not include axioms (they are not needed for the demo), we could add axioms such as:

- Names are less than 51 characters long.
- Type is one of the following: `ppl`, `summit`, `reserve`, ...

The SLANG and C++ GNIS specifications are connected by the SLANG-to-C++ morphism shown in Figure 11.

The final, highest level of wrapping is the SLANG specification `EXT-GNIS`, shown in Figure 12, which imports `GNIS` and `GEOGRAPHIC-COORDINATES`, an ontology describing representation-independent global locations. `EXT-GNIS` extends `GNIS` by defining `gnis-geoloc`, an operation to access the location of a GNIS record abstractly (as a `GeoLoc`), rather than concretely (as latitude and longitude represented by natural numbers).

5.1.2 Searching GNIS files

This part of the wrapper simply exports functionality from the low-level interface to the high-level interface, shown in Figure 13, changing languages from C++ to SLANG. It does less than the GNIS record interface, because it neither adds functionality nor changes representation. It exports the operations `find-first-gnis`, which finds the first GNIS record satisfying a predicate, and `find-all-gnis`, which finds all GNIS records satisfying a predicate. These operations retrieve records from the GNIS database.

5.1.3 Ontology operations

As noted above, the high-level interface includes `GEOGRAPHIC-COORDINATES`, an ontology describing locations on the globe. The wrapper must interpret the operations of this ontology if they are to be used in computation. Fortunately, an interpretation is available from the ontology library. Wrappers typically include part of the ontology library in order to describe the high-level interface in abstract terms. For example, the high-level interface to GNIS records uses abstract locations, while the low-level interface uses natural numbers.

```
(defun generate-gnis-dbi ()
  (gdbi-packed-text
   "gnis"
   '((name              string      0 51)
     (type              string     51 10)
     (county            string     61 32)
     (state             string     93 17)
     (latitude-degrees  nat       110  2)
     (latitude-minutes  nat       112  2)
     (latitude-seconds  nat       114  2)
     (latitude-ns       character 116  1)
     (longitude-degrees nat       117  3)
     (longitude-minutes nat       120  2)
     (longitude-seconds nat       122  2)
     (longitude-ew      character 124  1)
     (elevation         nat       125  6)
     )))
```

Figure 7: GNIS record description

```
class gnis
{ public:
  string    name;
  string    type;
  string    county;
  string    state;
  nat       latitude_degrees;
  nat       latitude_minutes;
  nat       latitude_seconds;
  character latitude_ns;
  nat       longitude_degrees;
  nat       longitude_minutes;
  nat       longitude_seconds;
  character longitude_ew;
  nat       elevation;
};
```

Figure 8: Fragment of GNIS C++ wrapper code

```
c-spec GNIS is
  import SLANG-BASE
  file "./gnis.cc"

  sort gnis

  op gnis-name              : gnis -> string
  op gnis-type              : gnis -> string
  op gnis-county            : gnis -> string
  op gnis-state             : gnis -> string
  op gnis-latitude-degrees  : gnis -> nat
  op gnis-latitude-minutes  : gnis -> nat
  op gnis-latitude-seconds  : gnis -> nat
  op gnis-latitude-ns       : gnis -> character
  op gnis-longitude-degrees : gnis -> nat
  op gnis-longitude-minutes : gnis -> nat
  op gnis-longitude-seconds : gnis -> nat
  op gnis-longitude-ew      : gnis -> character
  op gnis-elevation         : gnis -> nat

 end-c-specification
```

Figure 9: GNIS C++ specification

```
spec GNIS is
  sort gnis

  op gnis-name              : gnis -> string
  op gnis-type              : gnis -> string
  op gnis-county            : gnis -> string
  op gnis-state             : gnis -> string
  op gnis-latitude-degrees  : gnis -> nat
  op gnis-latitude-minutes  : gnis -> nat
  op gnis-latitude-seconds  : gnis -> nat
  op gnis-latitude-ns       : gnis -> char
  op gnis-longitude-degrees : gnis -> nat
  op gnis-longitude-minutes : gnis -> nat
  op gnis-longitude-seconds : gnis -> nat
  op gnis-longitude-ew      : gnis -> char
  op gnis-elevation         : gnis -> nat

end-spec
```

Figure 10: GNIS SLANG specification

```
spec-to-c-interpretation
  GNIS-to-GNIS : GNIS => GNIS is
  mediator GNIS
  dom-to-med
  sort-rules { gnis -> gnis }
  op-rules
  { gnis-name              -> gnis-name,
    gnis-type              -> gnis-type,
    gnis-county            -> gnis-county,
    gnis-state             -> gnis-state,
    gnis-latitude-degrees  -> gnis-latitude-degrees,
    gnis-latitude-minutes  -> gnis-latitude-minutes,
    gnis-latitude-seconds  -> gnis-latitude-seconds,
    gnis-latitude-ns       -> gnis-latitude-ns,
    gnis-longitude-degrees -> gnis-longitude-degrees,
    gnis-longitude-minutes -> gnis-longitude-minutes,
    gnis-longitude-seconds -> gnis-longitude-seconds,
    gnis-longitude-ew      -> gnis-longitude-ew,
    gnis-elevation         -> gnis-elevation }
  cod-to-med identity-morphism
```

Figure 11: GNIS SLANG-to-C++ interpretation

```
spec EXT-GNIS is
  import GNIS, GEOGRAPHIC-COORDINATES

  op gnis-latitude : Gnis -> Angle
  definition of gnis-latitude is
    axiom (equal (gnis-latitude g)
    (dms-dir-to-lat
     (gnis-latitude-degrees g)
     (gnis-latitude-minutes g)
     (gnis-latitude-seconds g)
     (gnis-latitude-ns g)))
  end-definition

  op gnis-longitude : Gnis -> Angle
  definition of gnis-longitude is
    axiom (equal (gnis-longitude g)
   (dms-dir-to-lng
    (gnis-longitude-degrees g)
    (gnis-longitude-minutes g)
    (gnis-longitude-seconds g)
    (gnis-longitude-ew g)))
  end-definition

  op gnis-geo-loc : Gnis -> Geo-Loc
  definition of gnis-geo-loc is
    axiom (equal (gnis-geo-loc g)
   (geo-loc
    (gnis-latitude g)
    (gnis-longitude g)))
  end-definition

end-spec
```

Figure 12: Top-level SLANG interface specification for GNIS

```
spec SCAN-GNIS-FILE is
  import LIST-of-GNIS

  sort Gnis-Predicate
  sort-axiom Gnis-Predicate = Gnis -> Boolean

  sort Gnis?
  sort-axiom Gnis? = Gnis + ()

  op find-first-gnis : String, Gnis-Predicate -> Gnis?
  op find-all-gnis   : String, Gnis-Predicate -> List-of-Gnis

end-spec
```

Figure 13: A SLANG interface specification with operations for accessing records from a GNIS file

5.2 Wrapping the DEM database

The DEM database contains elevation data, stored in a collection of files, one per one-degree by one-degree region. Naturally, each region begins and ends on a degree boundary, and its sides are parallel to the equator and the meridians. Our convention is to name each file by the latitude and longitude of its southwest corner, for example `38N-122W`.

For the US (except Alaska), each file contains elevations at a regular spacing of three arc seconds. That is, each file contains a 1201×1201 matrix of elevations. This matrix is stored as a sequence of columns from west to east; elevations within each column are stored from south to north. Elevations are represented in meters above mean sea level.

Each DEM file is stored in text format (rather than binary) and occupies about 9 mb; however `gzip` compresses it to about 1 mb. The continental US is tiled by approximately 1000 one-degree blocks; thus, the entire DEM database occupies about 1 gb when compressed.

The DEM wrapper is far more complex than the GNIS wrapper because it adds significant functionality. Its low-level interface centers on the operation

```
op dem1-column-in-file : String, Nat -> Dem1-Column
```

which retrieves from the file named by the first argument, the column of elevations at the position specified by the second argument.

The DEM wrapper's high-level interface centers on

```
op dem1-rectangle : Geo-Rect -> Dem1-Rectangle
```

which maps a geographic region, specified by a rectangle, to the matrix of elevation data for the region. Thus, the wrapper must patch together data from several one-degree rectangles to return a result.

The wrapper changes languages from C++ to SLANG and also changes the representation of regions and elevation matrices:

Data type	Low-level representation	High-level representation
Region	File name, column number	Abstract region
Elevations	Fixed-size column	Variable-size rectangle

6 Patching Multiple Representations

An important part of mediation is the reconciliation of different representations of the same information, an operation called "patching" in mathematics. The idea is to go from local descriptions and transition functions on overlapping parts to a global description, a recurring theme in the study of manifolds, bundles and sheaves [Mac Lane and Moerdijk 92, Steenrod 51].

6.1 An Analogy between Information Integration and Manifolds

Manifolds are generally constructed by patching or sewing together smaller parts. For an n-dimensional manifold, each part is characterized by an isomorphism (homeomorphism) into an open subset of $\mathbf{R}^n$, the n-dimensional real vector space. Such an isomorphism is called a *chart*, or a local coordinate system. The entire manifold is covered by a collection—called an *atlas*—of (possibly overlapping) charts. Wherever the charts overlap, there are *transition* functions which map one local coordinate system into another. In fact, a manifold is completely determined by the given subsets of $\mathbf{R}^n$ and the transition functions: it is obtained as the union of the subsets with points related by transition functions identified. The process is abstractly depicted in Figure 14. A global description may not exist for a manifold; it is generally unnecessary because operations on manifolds can be reduced to operations using the local coordinate systems and transition functions.

The situation is similar with information integration. We are given a collection of information sources. These may overlap in the sense that two sources may represent the same information differently: we thus need representation conversion functions. If we treat the individual information sources as local coordinate systems and the representation conversions as transition functions, then we can patch the sources together to obtain an integrated information source, the manifold.

6.2 An Example of Patching: Multiple Representations of Angles

We will illustrate patching using the example of angles (e.g., latitudes and longitudes) which generally have different representations in different databases. For example, in the GIS mediator (see Section 7.3, also Section 5), latitudes and longitudes are represented as arc seconds in the DEM (elevation grids) database, as degrees, minutes and seconds in the GNIS (geographic names) database, and as decimal degrees in ArcView (a map display application). Note that we can convert between these representations only if they represent the same abstract concept, in this case, angles. Thus, two databases can interoperate only if they have a shared ontology. This shared ontology resides in the mediator.

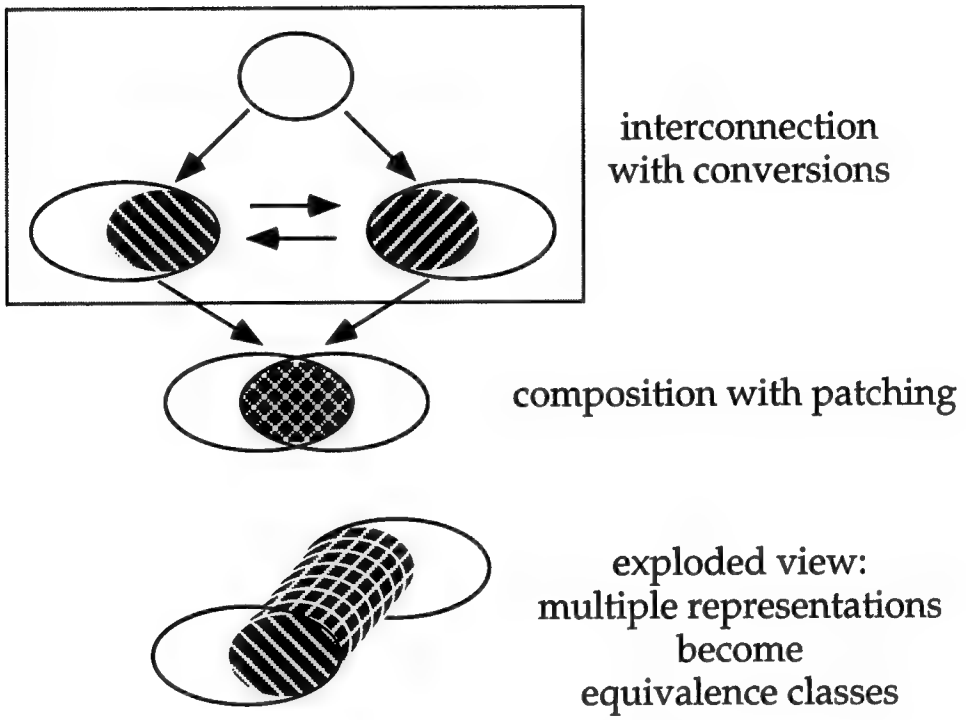

Figure 14: Abstract view of patching

Such a situation is depicted in Figure 15. ANGLE is an abstract theory of angles. The outer two morphisms from ANGLE are two representations of angles, as arc seconds and as degrees, minutes and seconds. The middle morphism from ANGLE is a dual representation obtained by combining the two representations using conversion functions.

A simplified specification for angles is shown in Figure 16 (a real specification would include many more operations). The specification states that angles form an abelian (i.e., commutative) group under the operation of addition.

Next, Figures 17 and 18 show two representations of angles. Each of these specifications provides a concrete representation of angles together with definitions of the required operations on angles.

Figures 19 and 20 show a specification which combines the two representations. First, conversion functions are defined between the two representations (`sec-to-dms`, `dms-to-sec`). These form an isomorphism between the representations. Next, a new type is constructed as the quotient under this isomorphism of the disjoint union of the two representations. In other words, elements of this new type are tagged versions of elements from either representation, and two elements are equal if they are either isomorphic or are equal in one of the two representations. Finally, all the required operations on angles are defined on this new type by a case analysis which dispatches to the corresponding operations on one of the two representations, inserting conversions where necessary.

A simpler (but less general) construction is possible if we choose one of the representations to be primary and convert all other representations into the primary one. This is the method we adopted in the GIS mediator.

6.3 Patching as a Composition Operator

In the example of Section 6.2 above, it is clear that the process of creating a specification for the dual representation is fairly canonical, with the only creative part being the definition of the isomorphism between the two representations. We are planning to add a new composition operator to SPECWARE to handle some of the details of the construction described above.

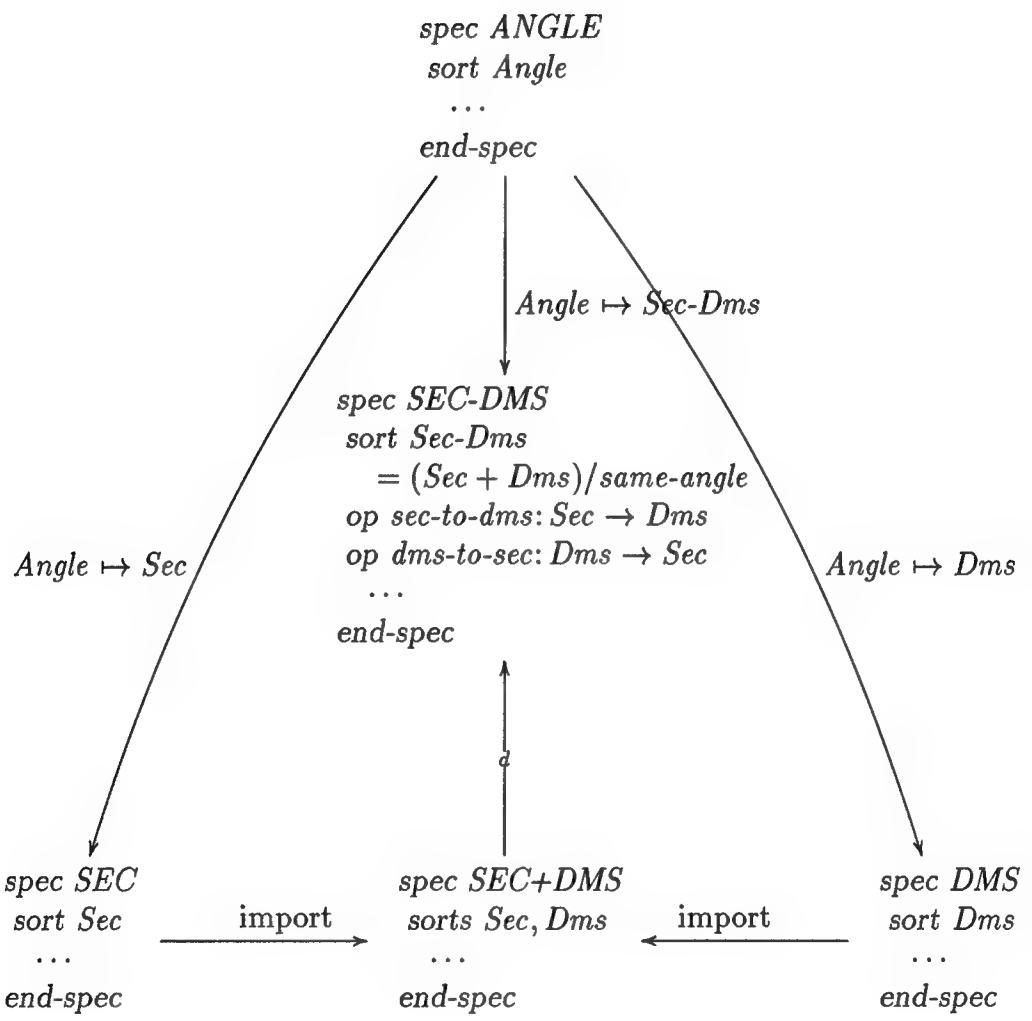

Figure 15: An example of patching: multiple representations of angles

```
spec ANGLE is
  sort Angle
  const zero : Angle
  op plus : Angle, Angle -> Angle
  op neg : Angle -> Angle
  % Abelian group axioms
  axiom associativity is
        (fa (x : Angle y : Angle z : Angle)
         (equal (plus x (plus y z)) (plus (plus x y) z)))
  axiom additive-identity is
        (fa (x : Angle)
         (equal (plus x zero) x))
  axiom additive-inverse is
        (fa (x : Angle)
         (equal (plus x (neg x)) zero))
  axiom commutativity is
        (fa (x : Angle y : Angle)
         (equal (plus x y) (plus y x)))
end-spec
```

Figure 16: A specification of some properties of angles (e.g., for use as latitudes or longitudes)

```
spec SEC is
  sort Sec
  sort-axiom Sec = (Nat, Boolean)

  const zero : Sec
  definition of zero is
    axiom (equal zero <0 true>)
  end-definition

  op plus : Sec, Sec -> Sec
  definition of plus is
    axiom (implies
           (equal s1 s2)
           (equal (plus <x s1> <y s2>) <(plus x y) s1>))
    axiom (implies
           (and (geq x y) (not (equal s1 s2)))
           (equal (plus <x s1> <y s2>) <(minus x y) s1>))
    axiom (implies
           (and (lt x y) (not (equal s1 s2)))
           (equal (plus <x s1> <y s2>) <(minus y x) s2>))
  end-definition

  op neg : Sec -> Sec
  definition of neg is
    axiom (equal (neg <x sign>) <x (not sign)>)
  end-definition
end-spec
```

Figure 17: A representation of angles as signed natural numbers (intended to be arc seconds)

```
spec DMS is
  sort Dms
  sort-axiom Dms = (Nat, Nat, Nat, Boolean)

  const zero : Dms
  definition of zero is
    axiom (equal zero <0 0 0 true>)
  end-definition

  op plus : Dms, Dms -> Dms
  definition of plus is
    % omitted
  end-definition

  op neg : Dms -> Dms
  definition of neg is
    axiom (equal (neg <d m s sign>) <d m s (not sign)>)
  end-definition
end-spec
```

Figure 18: A representation of angles as signed triples of natural numbers (intended to be degrees, minutes and seconds)

```
spec SEC-DMS is
  import SEC, DMS

  op sec-to-dms : Sec -> Dms
  definition of sec-to-dms is
    axiom (equal (sec-to-dms <x sign>)
           ((lambda (mins secs)
              <(div mins 60) (rem mins 60) secs sign>)
            (div x 60) (rem x 60)))
  end-definition

  op dms-to-sec : Dms -> Sec
  definition of dms-to-sec is
   axiom (equal (dms-to-sec <degs mins secs sign>)
          (<(plus (times 60 (plus (times 60 degs) mins)) secs) sign>))
  end-definition

  % sec-to-dms and dms-to-sec are inverses, i.e., an isomorphism
  theorem (equal (sec-to-dms (dms-to-sec x)) x)
  theorem (equal (dms-to-sec (sec-to-dms x)) x)

  sort Sec-Dms
  sort-axiom Sec-Dms = (Sec + Dms) / same-angle

  op same-angle : (Sec + Dms), (Sec + Dms) -> Boolean
  definition of same-angle is
    axiom (implies
           (equal x y)
           (same-angle x y))
    axiom (iff
           (same-angle ((embed 1) x) ((embed 2) y))
           (equal (sec-to-dms x) y))
    axiom (iff
           (same-angle ((embed 2) y) ((embed 1) x))
           (equal (dms-to-sec y) x))
  end-definition
```

Figure 19: Patching of two representations of angles (part 1)

```
  const zero : Sec-Dms
  definition of zero : Sec-Dms is
    axiom (equal zero ((quotient same-angle) ((embed 1) zero)))
  end-definition

  op neg : Sec-Dms -> Sec-Dms
  definition of neg : Sec-Dms -> Sec-Dms is
    axiom (equal
           (neg ((quotient same-angle) ((embed 1) x)))
           ((quotient same-angle) ((embed 1) (neg x))))
    axiom (equal
           (neg ((quotient same-angle) ((embed 2) y)))
           ((quotient same-angle) ((embed 2) (neg y))))
  end-definition

  op plus : Sec-Dms, Sec-Dms -> Sec-Dms
  definition of plus : Sec-Dms, Sec-Dms -> Sec-Dms is
   axiom (equal
          (plus ((quotient same-angle) ((embed 1) x1))
                ((quotient same-angle) ((embed 1) x2)))
          ((quotient same-angle) ((embed 1) (plus x1 x2))))
   axiom (equal
          (plus ((quotient same-angle) ((embed 1) x))
                ((quotient same-angle) ((embed 2) y)))
          ((quotient same-angle) ((embed 1) (plus x (dms-to-sec y)))))
   axiom (equal
          (plus ((quotient same-angle) ((embed 2) y))
                ((quotient same-angle) ((embed 1) x)))
          ((quotient same-angle) ((embed 1) (plus (dms-to-sec y) x))))
   axiom (equal
          (plus ((quotient same-angle) ((embed 2) y1))
                ((quotient same-angle) ((embed 2) y2)))
          ((quotient same-angle)
           ((embed 1) (plus (dms-to-sec y1) (dms-to-sec y2)))))
  end-definition
end-spec
```

Figure 20: Patching of two representations of angles (part 2)

7 Demonstrations

This section describes three simple demonstrations of Kestrel's technology for mediator construction. These demonstrations are the first in a sequence of prototypes that we expect will lead to the rapid construction of realistic mediators. The demonstrations are:

Scheduling: We combine a database of movements in TPFDD format (personnel and material to be transported between ports) and the GEOLOC database (port locations). We select movements based on distance and type. This demonstration outlines our basic technology for building mediators and was written for use with schedulers developed using the KIDS algorithm synthesis system.

SQL: We show how to model the SQL query language in SPECWARE, interface to an external SQL server, and reason about SQL queries. This demonstration shows techniques useful for wrapping and reasoning about external, language-based servers.

GIS: We combine the GNIS database (names and places in the US), the DEM database (digital elevation data), and ArcView (a COTS tool for displaying GIS data). Given the name of a place in the US, we find its location, extract a rectangular region of elevation data around it, select the names and locations of all places within the region, and send all data to ArcView for display. In essence, **GIS** is a more complex version of **Scheduling**.

7.1 Scheduling demonstration

Figure 21 shows the overall architecture of the scheduling mediator. This mediator extracts transportation domain data for use in the schedulers developed using KIDS, Kestrel's algorithm synthesis system [Smith 90].

First, each database is wrapped by a specification describing its interface. The movement database is read via an operation `select` that returns all the movements satisfying a predicate, while the the GEOLOC database is read via an operation `port-location` that maps ports to locations. The two database specifications import fragments of a global ontology describing ports, locations, movements, and other concepts associated with them, such as distance and time.

Second, the two interface specifications are glued together, sharing whatever parts of the ontology they have in common. This ontological commonality allows us to relate the two databases; without it, we could not. The result is a combined database.

Finally, we refine the application interface into the interface of the combined database. Here, we specify an operation that returns all the movements that must travel at least a certain distance. This function reads movement records from the first database and computes how far they must travel via the second. It uses operations for computing distance from the common ontology, refinements of which are available from the library.

Figure 22 shows a more detailed view of the scheduling mediator, indicating the components of the common ontology and database interfaces. The two database interfaces share `Basics`, a domain-independent ontology about mathematical and physical concepts, and `Port`, a domain-specific ontology about ports. Ports are the single domain-specific link between the two databases: each movement specifies its source and destination ports, and the GEOLOC database maps ports to locations. The `GeoLoc` specification describes single records, while `GeoLoc DB` describes the entire database, and similarly for `Movement` and `Movement DB`.

Figure 23 shows a sample interaction with the scheduling mediator, which has been refined to Common Lisp code. In it, we ask for all movements that require travel of at least 7000 miles. This query requires access to both databases.

Although the mediator performs a trivial task, we believe that the architecture we have developed will allow us to scale to much larger examples in a disciplined way. Specifically, an ontology library, refined to executable code, provides the necessary leverage to connect multiple databases and pose queries over the result. Adding new queries is easy because the *ontologies* are rich and already contain the necessary concepts. Adding new databases is also easy, because the ontology *library* is rich and already contains the necessary ontologies. The real work lies in extending the ontology library when a new domain is encountered.

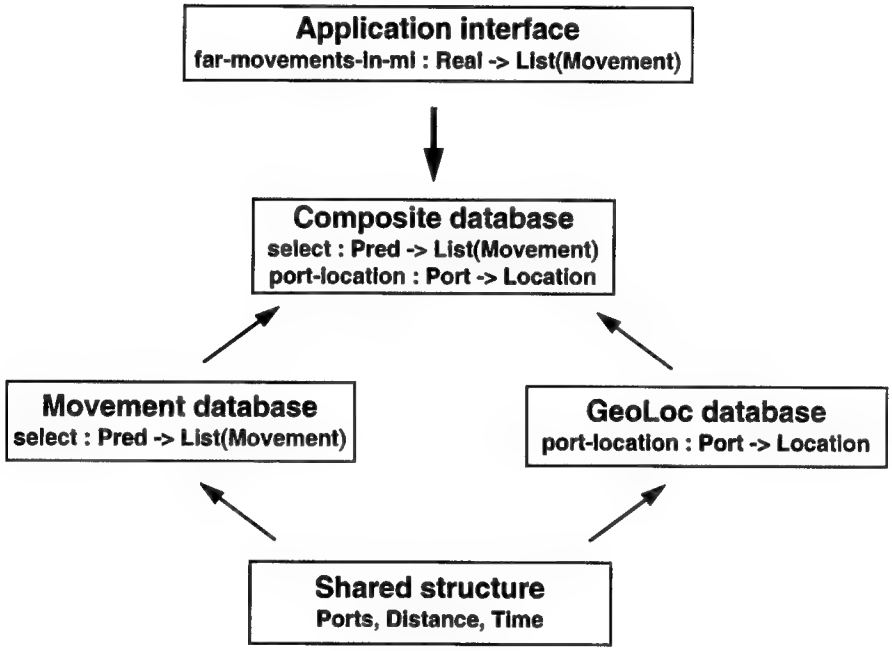

Figure 21: Scheduling mediator overall architecture

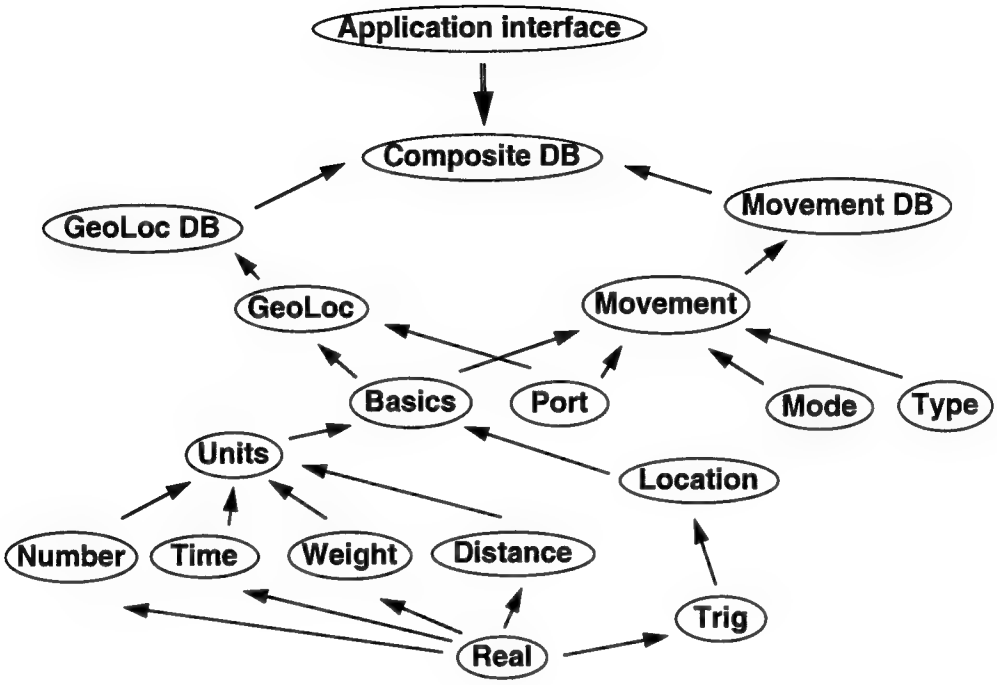

Figure 22: Scheduling mediator detailed architecture

```
.> (far-movements-in-mi 7000)

((OVR-MOVEMENT 6 86400 172799 PQWY FTZH 0 SEA)
 (PAX 29 0 86399 SCEY XBGX 0 AIR)
 (OUT-MOVEMENT 13 0 86399 SCEY XBGX 0 SEA))

.> (distance-in-mi '(OVR-MOVEMENT 6 86400 172799 PQWY FTZH 0 SEA))

7853.369779442932
```

Figure 23: Interacting with the scheduling mediator

7.2 SQL demonstration

Suppose we want to specify a formal semantics for an external SQL server with which we communicate using text strings. How can we specify that the server is actually answering the queries we have in mind, and not some other queries chosen at random?

This problem requires some thought to appreciate, so let's consider first a simple server for addition and multiplication. This example is nonsensical because these operations can be computed quickly locally, but the problem of communicating with the server via text remains. Suppose we send the server the string `(+ 3 5)`. How can we reason formally that we will receive `8` as an answer? After all, we can't add numbers in textual form, and parsing is inconvenient. Still, we need a connection between the textual representation and the abstract, numerical representation. Without this connection, the server could return `9` instead of `8` and we wouldn't be any wiser.

7.2.1 Denotational semantics

The theory of denotational semantics provides a simple answer. Let's consider several models of the specification

```
spec ARITHMETIC
  sort Num
  op num : Nat -> Num
  op +   : Num, Num -> Num
  op *   : Num, Num -> Num
end-spec
```

which describes simple arithmetic expressions. We can represent models as refinements to other specifications. Figure 24 shows a model in which the arithmetic operators build text strings. For instance, `(+ (* (num 3) (num 8)) (num 2))` evaluates to `"(+ (* 3 8) 2)"`. Figure 25 shows a model in which the operators actually perform arithmetic, so that the same expression evaluates to `26`. Figure 26 shows a model in which the operators construct abstract syntax trees, so that we obtain a tree with two nodes, for `*` and `+`, and three leaves, for `3`, `8`, and `2`. We call these three models *concrete*, *semantic*, and *abstract*, respectively.

In general, the semantic model is used for reasoning and actually performing arithmetic, while the concrete model is used for communication with the external server. The abstract model is used to connect the other two, our primary goal.

Each of these refinements is called an *algebra* over the signature given by the above specification. An *algebra homomorphism* over the same signature is an operation satisfying the laws shown in Figure 27. The algebras over a fixed signature form a category, with homomorphisms as arrows, since the composition of two homomorphisms is a homomorphism.

```
spec CONCRETE is
  import ARITHMETIC
  sort-axiom Num = String

  axiom (equal (num n) (nat-to-string n))
  axiom (equal (+ a b) (string-append "(+ " a " " b ")"))
  axiom (equal (* a b) (string-append "(* " a " " b ")"))
end-spec
```

Figure 24: Concrete syntax

```
spec SEMANTIC is
  import ARITHMETIC
  sort-axiom Num = Nat

  axiom (equal (num n) n)
  axiom (equal (+ a b) (plus a b))
  axiom (equal (* a b) (times a b))
end-spec
```

Figure 25: Semantics

```
spec ABSTRACT is
  import ARITHMETIC
  sort-axiom Num = Nat + (Num, Num) + (Num, Num)

  axiom (equal (num n) ((embed 1) n))
  axiom (equal (+ a b) ((embed 2) a b))
  axiom (equal (* a b) ((embed 3) a b))
end-spec
```

Figure 26: Abstract syntax

```
spec ARITHMETIC-HOMOMORPHISM is
  import
    translate ARITHMETIC by
      Num -> Num1, num -> num1, + -> +1, * -> *1 ,
    translate ARITHMETIC by
      Num -> Num2, num -> num2, + -> +2, * -> *2

  op h : Num1 -> Num2

  axiom (equal (h (num1 n)) (num2 n))
  axiom (equal (h (+1 a b)) (+2 (h a) (h b)))
  axiom (equal (h (*1 a b)) (*2 (h a) (h b)))

end-spec
```

Figure 27: Arithmetic homomorphism

An object of a category is called *initial* if there is a unique arrow from it to any other object. In fact, the abstract model is an initial algebra for the arithmetic signature, and the unique homomorphism is given by evaluating expressions, represented as abstract syntax trees, over the target algebra. Thus, the homomorphism to the concrete algebra produces concrete syntax from abstract, while the homomorphism to the semantic algebra produces actual numbers. We call the former `rep`, for representation, and the latter `den`, for denotation.

We interact with the external server by extending the `CONCRETE` specification with an operation

```
concrete-value : Num -> Nat
```

which we implement by sending the text string to the arithmetic server and returning the result as a number.

Similarly, we extend the `SEMANTIC` specification with an operation

```
semantic-value : Num -> Nat
```

which is actually the identity, since `Num` and `Nat` are identical.

Finally, it remains to be seen how the abstract model allows us to connect the concrete and semantic models. We extend a specification containing all of the above with the single axiom

```
(concrete-value (rep as)) = (semantic-value (den as))
```

This axiom states that the values produced by the server are the same as those produced by the semantic "reference" implementation. For example, we can reason as follows:

```
  (concrete-value      (c+ (c-num 3) (c-num 5)))
= (concrete-value (rep (a+ (a-num 3) (a-num 5))))
= (semantic-value (den (a+ (a-num 3) (a-num 5))))
= (semantic-value      (s+ (s-num 3) (s-num 5)))
= 8
```

We write `a-`, `c-` and `s-` to distinguish the three algebras. The first and last steps are justified because `rep` and `den` are homomorphisms. The middle step is justified by the axiom.

The axiom also allows us to transfer algebraic laws to the external server. For example, we can show commutativity of addition:

```
  (concrete-value      (c+ (rep a1) (rep a2)))
= (concrete-value (rep (a+ a1 a2)))
= (semantic-value (den (a+ a1 a2)))
= (semantic-value      (s+ (den a1) (den a2)))
= (semantic-value      (s+ (den a2) (den a1)))
= (semantic-value (den (a+ a2 a1)))
= (concrete-value (rep (a+ a2 a1)))
= (concrete-value      (c+ (rep a2) (rep a1)))
```

The reasoning steps are essentially the same as before. We rewrite from concrete to semantic, apply the relevant law, and reverse our steps. The explicit use of abstract syntax allows us to quantify over all arithmetic expressions `a1` and `a2`.

Of course, we could begin with the axiom

```
(concrete-value (c+ (rep a1) (rep a2))) =
(concrete-value (c+ (rep a2) (rep a1)))
```

and spare ourselves the intermediate reasoning, but the point is that the machinery we have constructed immediately transfers *all* axioms from semantic to concrete in a single step. Without it, we would need to transfer each axiom individually.

Note that the simplified axiom

```
(c+ (rep a1) (rep a2)) = (c+ (rep a2) (rep a1))
```

is simply not true, since the two sides are different arithmetic expressions, even though their values are the same.

7.2.2 *Semantics of SQL*

Now that we have understood the general approach, let's see how it applies to SQL. First we need to specify the SQL language itself. Leaving out some details, the usual syntax for the SQL `select` statement is roughly:

```
SELECT [DISTINCT] field+
 [INTO table]
 [FROM table+]
 [WHERE condition+]
 [GROUP BY [ALL] column+]
   [HAVING condition+]
 [ORDER BY column [DESC]+]
```

Depending on which options are specified, `select` can perform rather complex combinations of joining, filtering, grouping, accumulating, sorting, and projecting. We have chosen to decompose `select` into five simpler operations:

```
basic-select
  : Distinct?, Projection, Tables, Where?, Order? -> Table
aggregate-select
  : Aggregate, Tables, Where? -> Element
group-select
  : Projection, Tables, Where?, Same?, Having?, Order? -> Table
group-summary-select
  : Summary, Tables, Where?, Same?, Having?, Order? -> Table
group-aggregate-select
  : Aggregate, Tables, Where?, Same?, Having? -> Element
```

These functions differ subtly according to what kind of data is returned and what operations are applied to it:

`basic-select` joins several tables, filters rows according to a predicate, projects onto several fields, sorts the result, and possibly eliminates duplicate rows.

`aggregate-select` joins several tables, filters rows according to a predicate, and applies an aggregation function to the resulting table such as counting, summing, or averaging,

`group-select` joins tables, filters rows, groups rows into equivalence classes, filters equivalence classes, picks a representative from each class, applies a projection function, and sorts the result.

`group-summary-select` is like `group-select` but instead of choosing arbitrarily from each equivalence class, it applies a more given summary function.

`group-aggregate-select` joins tables, filters rows, groups rows into equivalence classes, filters equivalence classes, picks a representative from each class, and applies an aggregation function to the resulting table.

Even given a general understanding of these operations, there are several choices for their exact semantics. For example, should we project before or after we sort? Should we project before or after we group into equivalence classes?

To specify the semantics of these operations, we proceed as in the last section. We will skip over the details of SQL algebras, but, as before, we have three algebras, concrete syntax, abstract syntax, and semantics, and two homomorphisms `rep` and `den`. The concrete algebra allows interaction with the SQL server, while the semantic algebra provides actual semantic content.

Figure 28 shows the definition of the SQL semantic algebra. Each operation is specified as a specific combination of joining, filtering, grouping, accumulating, sorting, and projecting. Each of these mathematical operations is easily specified via the appropriate laws. Thus, the semantics of SQL is given by reduction to these more basic operations, and, as before, the homomorphisms `rep` and `den` allow us to reason about the behavior of the server.

Figures 30 through 34 show several English language queries and their formulations using the SQL operators. These examples refer to a sample publishing database of books, authors, and publishers, whose schema is shown in Figure 29. Each Figure shows the query, the columns of the joined tables used to answer it, and the query formulation. For simplicity, columns are referenced by number.

The relationship between the query and the operator chosen to answer it is not always obvious, because the database is organized around different concepts than the query. A theorem prover may be used to search for a low-level formulation of a high-level query, assuming that the database is wrapped in a formal theory.

```
(equal (basic-select
          distinct-f projection table where? order?)
 (distinct-f
  (sort-table
   (map-table projection
    (filter-table table where?))
   order?)))

(equal (aggregate-select aggregate table where?)
 (aggregate (filter-table table where?)))

(equal (group-select
          projection table where? same? having? order?)
 (group-summary-select
  (lambda (neT) (projection (first-row neT)))
  table where? same? having? order?))

(equal (group-summary-select
          summary table where? same? having? order?)
 (sort-table
  (summarize-partition summary
   (filter-partition
    (partition-table (filter-table table where?) same?)
    having?))
  order?))

(equal (group-aggregate-select
          aggregate table where? same? having?)
 (aggregate
  (summarize-partition first-row
   (filter-partition
    (partition-table (filter-table table where?) same?)
    having?))))
```

Figure 28: SQL semantics

Table	Field 0	Field 1	Field 2	Field 3
Authors	Author Id	Name	Address	
Books	Book Id	Title	Pub Id	Price
Authors-Books	Author Id	Book Id		
Publishers	Pub Id	Name	Location	

Figure 29: Sample publishing database

```
%%  What are the titles and price's of Sue's books, sorted by title?

%%  0      1      2         3      4      5      6      7       8
%%  Au-Id, Name, Address, Au-Id, Bk-Id, Bk-Id, Title, Pub-Id, Price

(basic-select
 not-distinct                                  % Distinct?
 (lambda (r)                                   % Projection
   (make-row2 (column r 6) (column r 8)))
 (join authors (join authors-books books))     % Tables
 (lambda (r)                                   % Where?
   (and
    (equal "sue" (column r 1))
    (and (equal (column r 0) (column r 3))
         (equal (column r 4) (column r 5)))))
 (lambda (r1 r2)                               % Order?
   (elt-le? (column r1 0) (column r2 0))))
```

Figure 30: Query using `basic-select`

```
%%  If I buy one of each of Bob's books, how much do I have to spend?

%%  0      1      2         3      4      5      6      7        8
%%  Au-Id, Name, Address, Au-Id, Bk-Id, Bk-Id, Title, Pub-Id, Price

(aggregate-select
 (lambda (table)                                    % Aggregate
   (sum-table (lambda (r) (column r 8)) table))
 (join authors (join authors-books books))          % Tables
 (lambda (r)                                        % Where?
   (and
    (equal "bob" (column r 1))
    (and (equal (column r 0) (column r 3))
         (equal (column r 4) (column r 5))))))
```

Figure 31: Query using aggregate-select

```
%%  Which books have more than one author?

%%  0      1      2       3      4      5
%%  Bk-Id, Title, Pub-Id, Price, Au-Id, Bk-Id

(group-select
 (lambda (r)                                        % Projection
   (make-row1 (column r 1)))
 (join books authors-books)                         % Tables
 (lambda (r)                                        % Where?
   (equal (column r 0) (column r 5)))
 (lambda (r1 r2)                                    % Same?
   (equal (column r1 0) (column r2 0)))
 (lambda (table)                                    % Having?
   (not (elt-le? (count-table table) elt-1)))
 (lambda (r1 r2) true))                             % Order?
```

Figure 32: Query using group-select

```
%%  What is each publisher's average book price,
%%  sorted by publisher's name?

%%  0      1      2       3      4       5     6
%%  Bk-Id, Title, Pub-Id, Price, Pub-Id, Name, Location

(group-summary-select
 (lambda (table)                                    % Summary
   (make-row2
    (column (first-row table) 5)
    (avg-table (lambda (r) (column r 3)) table))
 (join books publishers)                            % Tables
 (lambda (r)                                        % Where?
   (equal (column r 2) (column r 4)))
 (lambda (r1 r2)                                    % Same?
   (equal (column r1 2) (column r2 2)))
 (lambda (table) true)                              % Having?
 (lambda (r1 r2)                                    % Order?
   (elt-le? (column r1 0) (column r2 0)))))
```

Figure 33: Query using group-summary-select

```
%%  How many books have at least one author living in Palo Alto?

%%  0      1     2       3      4
%%  Au-Id, Name, Address, Au-Id, Bk-Id

(group-aggregate-select
 count-table                                        % Aggregate
 (join authors authors-books)                       % Tables
 (lambda (r)                                        % Where?
   (and (equal (column r 0) (column r 3))
        (equal (column r 2) "Palo Alto")))
 (lambda (r1 r2)                                    % Same?
   (equal (column r1 4) (column r2 4)))
 (lambda (table) true))                             % Having?
```

Figure 34: Query using group-aggregate-select

7.3 GIS demonstration

The GIS mediator displays elevation data for a rectangular region around a given location. Figure 35 shows the results produced by the GIS mediator. After starting the ArcView tool, we execute a query script, which brings up a dialog box asking for the name of a place and the size of a rectangle around it, in arc seconds. The mediator then produces a display like the one shown in the figure. The mediator performs the following actions:

- It obtains the location of the place from the GNIS database.
- It obtains the elevation data from the DEM database.
- It obtains the names and locations of places within the rectangle from the GNIS database.
- It sends all this data to ArcView.

The GNIS and DEM formats were already described in Chapter 5 on wrappers. We first describe the ArcView interface, then the mediator itself.

7.3.1 ArcView

ArcView is a widely used visualization tool in the commercial GIS field. It is produced by Environmental Systems Research Institute (ESRI) of Redlands, CA, which also produces ArcInfo, the standard system for GIS computation. ArcView can display and query GIS data in both vector and raster formats and includes a scripting language called Avenue.

ArcView accepts data from the the mediator in two formats:

- a `.txt` file containing the names of places and their locations, shown in Figure 36, and
- a `.tiff` file containing the elevation data image in a standard format, together with a `.tiffw` file ("`w`" for "world"), shown in Figure 37, describing the size and geographic location of the `.tiff` file.

The `.txt` file format is straightforward, except that locations are represented in signed decimal degrees, while locations in the GNIS database are represented in degrees, minutes, seconds, and direction. The mediator converts between the two representations; for example, `372631N` becomes `37.441944`.

The mediator writes elevation data in `.raw-pgm` format; elevations are represented as decimal numbers, one per line in text format. To create a `.tiff` image, we run the

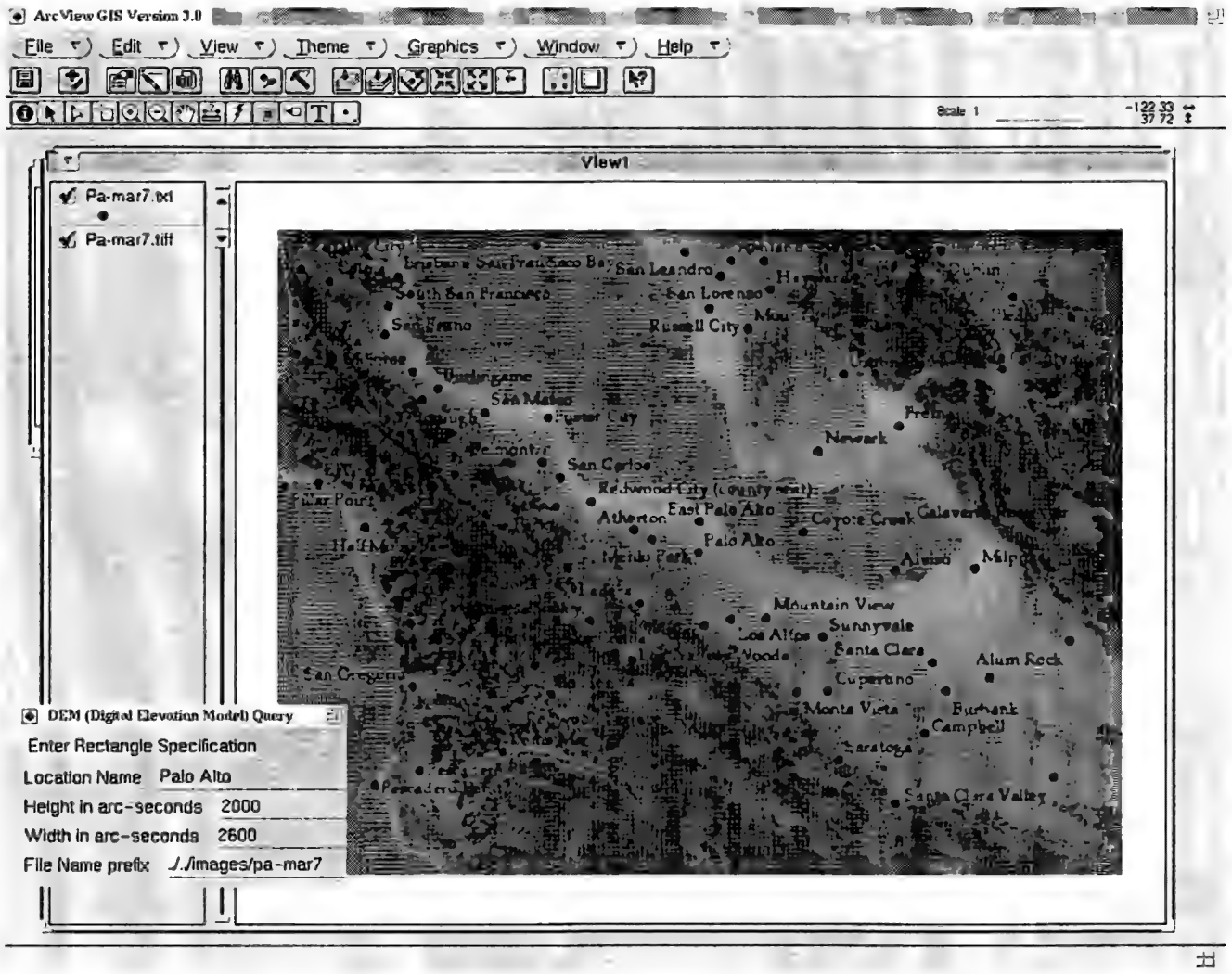

Figure 35: GIS mediator demonstration

shell script shown in Figure 38, which also colors the data according to a standard elevation color map. The mediator writes `.txt` and `.tiffw` files directly.

The `.tiffw` file informs ArcView where the `.tiff` image is located on the earth, what size it is, and how it oriented, so that ArcView can display it properly. The file gives an affine transformation from the image grid (with indices 0, 1, 2, ...) to the world coordinate system (in decimal degrees). Since DEM image data is parallel to the equator and meridians, no rotation or shear is necessary, so the second and third numbers are always zero. The first and fourth numbers then give the degrees per pixel in the horizontal and vertical directions; for 1-degree DEM files, they are always positive and negative three arc seconds, the standard DEM spacing. The final two numbers specify the location of the image's southwest corner.

7.3.2 GIS mediator structure

Figure 40 shows the top level specification of the GIS mediator. It imports the colimit of the diagram shown in Figure 39. The diagram indicates the overall structure of the mediator, while the final specification shows how the different elements are used in the mediation process.

The operation `gnis-dem-main` takes two strings and two numbers and produces an action, which is to write several files. Actions are discussed in the next section. The strings represent the name of a place to locate and a file name prefix for the files to be written. The numbers represent the height and width of the rectangle in arc seconds. The function finds the place, complains if it doesn't exist, and writes the relevant image, world, and text files.

The mediator is composed of three wrappers, glued together via a shared ontology, which ties them together conceptually. Without the shared ontology, the databases could not exchange data. The shared ontology is composed of three specifications, `Geographic-Rectangle`, which describes rectangular geographic regions, `List-of-Gnis`, which describes lists of GNIS records, and `Dem1-Rectangle`, which describes grids of elevation data. The morphisms in the diagram show which ontology components are required by which wrappers; careful consideration shows that these are the necessary and sufficient relationships between the wrappers.

```
Location, Latitude, Longitude
Woodside, 37.43, -122.253
Upper Crystal Springs Reservoir, 37.5094, -122.35
Union City, 37.5958, -122.018
Tunitas, 37.3817, -122.388
Sunnyvale, 37.3689, -122.035
...
```

Figure 36: ArcView `.txt` file

```
0.000833333
0
0
-0.000833333
-122.503
37.7197
```

Figure 37: ArcView `.tiffw` file

```
#!/bin/sh

main "$1" $2 $3 $4
scale < $4.raw-pgm > $4.pgmr
pnmflip -rotate90 $4.pgmr > $4.pgm
pgmtoppm -map ../utilities/256b.ppm $4.pgm > $4.ppm
pnmtotiff -none $4.ppm > $4.tiff
rm $4.pgmr $4.pgm $4.ppm
```

Figure 38: GIS mediator shell script

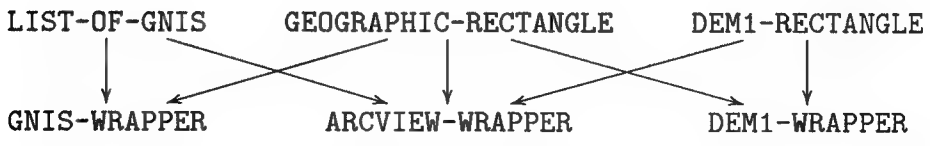

Figure 39: GIS mediator architecture

```
spec GNIS-DEM1-MAIN is
  import GNIS-DEM1-MAIN-import

  op gnis-dem-main : String, String, Distance, Distance -> Action
  definition of gnis-dem-main is
   axiom (equal (gnis-dem-main place file w h)
          (gnis-dem-aux1
           (find-first-gnis "ca-concise" < name-equal? place >)
           file w h))
  end-definition

  op gnis-dem-aux1 : Gnis?, String, Distance, Distance -> Action
  definition of gnis-dem-aux1 is
   axiom (equal (gnis-dem-aux1 ((embed 2) x) file w h)
          (write-string "Can't find place!\\n"))
   axiom (equal (gnis-dem-aux1 ((embed 1) g) file w h)
          (gnis-dem-aux2 file (geo-rect (gnis-geo-loc g) w h)))
  end-definition

  op gnis-dem-aux2 : String, Geo-Rect -> Action
  definition of gnis-dem-aux2 is
   axiom (equal (gnis-dem-aux2 file r)
          (seq-actions
           (write-pgm-file
            (concat-string file ".raw-pgm")
            (dem1-rectangle r))
           (seq-actions
            (write-world-file
             (concat-string file ".tiffw")
             r)
            (write-gnis-table
             (concat-string file ".txt")
             (find-all-gnis "ca-concise" < gnis-in? r >)))))
  end-definition

end-spec
```

Figure 40: Top level GIS mediator specification

7.3.3 *Actions*

As part of the GIS mediator, we have included a theory of *actions*. This theory allows us to mix imperative and functional programming styles. Normally, in a purely functional language such as SLANG or Haskell, computations can only return values; they cannot perform I/O operations, such as writing files. Needless to say, it is useful for mediators to write files! There are two solutions to this problem:

- Abandon purely functional languages.
- Treat I/O-performing computations as values.

In the former, we program in a mixed functional / imperative language such as Lisp or C, in which we can execute `printf("xyz")` or `(format t "xyz")`. In the latter, we use a purely functional language to construct *actions*. Actions are *values* that represent I/O-performing *computations*. For example, `(write-string "xyz")` is an action that, *when executed*, writes `"xyz"`.

Figure 41 shows the `ACTION` specification, which states that actions form a monoid under `null-action` and `seq-actions`; in other words, actions are sequences of more basic actions. The basic actions are writing natural numbers, characters, strings, and newlines. In addition, actions can be directed to files using `with-output-to-file`. For example, the action

```
(with-output-to-file "foo"
 (seq-actions
  (write-nat 43)
  (seq-actions
   (write-string "hello")
   write-newline)))
```

writes `"43hello"`, followed by a newline, to the file `foo`.

Functional programs can *return* actions, but they cannot *execute* them. Once returned, actions are executed by an *external* agency (the surrounding system), so that the functional computation remains free of side-effect. Actions are not executed unless they are sequenced into the final action returned at the end. In particular, an action that is constructed and then "thrown away" is never executed.

In our current implementation, all actions are executed at the end; however, we could alternatively interleave the execution of the functional program and the actions it returns. That is, we could run the functional program to produce an action, execute the action, then continue the program to produce more actions. This strategy would allow us to reclaim some of the space used to produce the initial actions, reducing the storage requirements of the program.

```
spec ACTION is
  sort Action

  op write-nat     : Nat    -> Action
  op write-char    : Char   -> Action
  op write-string  : String -> Action
  op write-newline : Action

  const null-action : Action
  op    seq-actions : Action, Action -> Action

  %% Actions form a monoid:
  axiom (equal (seq-actions a null-action) a)
  axiom (equal (seq-actions null-action b) b)
  axiom (equal (seq-actions a (seq-actions b c))
               (seq-actions (seq-actions a b) c))

  op with-output-to-file : String, Action -> Action
end-spec
```

Figure 41: Action specification

Compared to Lisp or C, actions in functional language are easier to reason about but less modular. For example, in SLANG it is always true that

```
(+ (f x) (f x)) = (* 2 (f x))
```

but, in a mixed language, `f` may print `"hello"`, in which case we obtain two hellos from the left side and only one from the right. Thus, in order to reason effectively, we need knowledge of which operations perform side-effects. In the functional language, there is only one kind of computation; side-effects are not allowed.

On the other hand, a mixed language is very convenient, because actions are implicit and implicitly sequenced. If we change a low-level function to print a message for debugging, we don't need to change every function that calls it to return an action. In other words, small conceptual changes require only small textual changes.

The long-standing debate between the functional and imperative schools is not yet over, although the issues involved are understood fairly well. It is clear that both sides have significant advantages and disadvantages; what remains is to find an acceptable synthesis between the two. One possibility is a type system that keeps track of which computations are imperative, yet allows them to be implicitly sequenced.

The basic idea behind actions was invented independently by several researchers:

- by the Algol 60 commitee, as pointed out later by John Reynolds, while designing a programming language,
- by John McCarthy, while reasoning about actions in planning,
- by Peter Henderson, while drawing graphical pictures in a functional language,
- by Eugenio Moggi, while structuring the denotational semantics of mixed languages.

Reynolds emphasized Algol's subtle distinction between *values* and *phrases* (i.e. *computations* or *actions*) [Reynolds 81]. For example, `5` is a value, while `2+3` and `1+4` are phrases. Both of these can be passed to and returned from functions. As above, this distinction allows Algol programs to be conceptually evaluated in two phases, a functional phase that yields an imperative program (an action), and an imperative phase that executes it.

McCarthy invented the *situation calculus* for reasoning about actions in blocks-world planning [McCarthy and Hayes 69]. The calculus includes expressions such as `puton(A, B)`, the action of putting block `A` on block `B`, and `result(a, s)`, the state obtained by performing action `a` in state `s`. McCarthy was probably the first author to represent actions as values.

Henderson invented a way to express graphical pictures in a functional language [Henderson 80]. He used combinators, such as horizontal and vertical juxtaposition, rotation, and reflection, to build complex pictures from simpler parts. None of the operators actually cause pictures to appear on the screen; they simply construct pictures from other pictures, as though pictures were values. When a picture is finally returned by a program, it is "magically" displayed on the screen.

Eugenio Moggi generalized actions using the category-theoretic concept of *monad* [Moggi 89]. Moggi showed how monads can represent many different types of imperative computation (state, I/O, nondeterminism, continuations, parallelism, exceptions, etcetera). Moggi's ideas were popularized by Philip Wadler [Wadler 92], and numerous papers have since been written on this topic in the functional programming community.

The GIS mediator uses actions to write the `.tiff`, `.tiffw`, and `.txt` files. Using actions, the handwritten main program for the mediator is very simple: extract the command line arguments, pass them to the SPECWARE-generated mediator, and perform the action that it returns. This action writes the files needed by ArcView, which displays the appropriate images.

8 Results and Future Plans

The results of our project so far indicate progress on all aspects of our approach: a formal specification and refinement process applied to the development of mediators.

Formal wrappers: A formal wrapper is a logical specification of the relevant functionality (interface) of an information source, together with a formal connection to the code which realizes it. In SPECWARE, a wrapper specification is a theory in higher-order logic, while the code realizing it can be in Lisp or C++.

We have constructed such wrappers for the GNIS (Geographic Name Information System) and DEM (Digital Elevation Model) databases from the US Geological Survey, and for simplified versions of the TPFDD (movement requirements) and GEOLOC (port locations) databases from the DARPA Planning Initiative. In addition, we have shown how to wrap an external SQL server.

Composition and patching: The wrapper (or interface) specifications mentioned above are structured, i.e., composed from smaller specifications. SPECWARE provides flexible ways of interconnecting and composing specifications, enabling rapid construction of interfaces using pre-existing libraries.[1]

Patching is the composition of specifications which represent a shared concept differently. Patching also applies to refinements: it is the composition of refinements which refine a shared concept differently. A simple version of patching is used in combining the GNIS and DEM databases to handle different representations of geographic coordinates.

Mediator generation: In the SPECWARE framework, mediator generation is the refinement/realization of the application interface in terms of the interfaces of the information sources. As already mentioned, SPECWARE supports the composition and patching of the interface specifications; this composition structure can be exploited to build a refinement from the application interface into the code connected to the wrappers.

The GIS demonstration (Section 7.3), which combines the GNIS and DEM databases, is an example. The application interface consists of a single kind of query: retrieve elevation and location information in a rectangular region around a given place. This function, after several steps of refinement, is finally realized by the basic functionality provided by the GNIS and DEM databases. Currently, this refinement process is manual.

[1] Of course, the building of useful libraries is a hard problem.

Future Plans

In subsequent work, we plan to automate the mediator development process and exploit this automation to construct larger examples.

System support for patching: The goal here is to provide a new composition operator, patching, in SPECWARE. Such an operator, in the basic case, would take two different refinements (representations) of a concept, together with conversion functions relating the two representations, and combine them into a single refinement (i.e., a dual representation). In other words, an implementation of patching will transparently handle multiple representations of concepts.

Mediator development automation: The process of refinement in SPECWARE is mostly manual because of its generality. Mediator development is more specific and stylized, so it is possible to automate some parts, e.g., the refinement to code of an interface specification built by composing and patching together wrapper specifications which have direct realizations as code.

We will also leverage any progress on SPECWARE which helps us to tackle larger mediation problems.

References

[Blaine and Goldberg 91]

BLAINE, L., AND GOLDBERG, A. DTRE – a semi-automatic transformation system. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 165–204.

[Gilham et al. 89]

GILHAM, L.-M., GOLDBERG, A., AND WANG, T. C. Toward reliable reactive systems. In *Proceedings of the 5th International Workshop on Software Specification and Design* (Pittsburgh, PA, May 1989).

[Henderson 80]

HENDERSON, P. *Funcional Programming: Application and Implementation.* Prentice-Hall, 1980, pp. 255–267.

[Lambek and Scott 86]

LAMBEK, J., AND SCOTT, P. J. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, Cambridge, 1986.

[Mac Lane and Moerdijk 92]

MAC LANE, S., AND MOERDIJK, I. *Sheaves in Geometry and Logic.* Springer-Verlag, New York, 1992.

[McCarthy and Hayes 69]

MCCARTHY, J., AND HAYES, P. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, B. Meltzer and D. Mitchie, Eds. Edinburgh University Press, Edinburgh, 1969.

[Meseguer 89]

MESEGUER, J. General logics. In *Logic Colloquium'87*, H.-D. Ebbinghaus et al., Eds. North-Holland, 1989, pp. 275–329.

[Moggi 89]

MOGGI, E. Computational lambda calculus and monads. In *IEEE Symposium on Logic in Computer Science* (Asilomar, CA, June 1989), pp. 14–23.

[Reynolds 81]

REYNOLDS, J. C. The essence of Algol. In *Algorithmic Languages*, de Bakker and van Vliet, Eds. North-Holland, Amsterdam, 1981, pp. 345–372.

[Smith 90]

SMITH, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (September 1990), 1024–1043.

[Srinivas and Jüllig 95]

SRINIVAS, Y. V., AND JÜLLIG, R. Specware:^{TM} Formal support for composing software. In *Conference on Mathematics of Program Construction* (Kloster Irsee, Germany, July 1995), B. Moeller, Ed., *Lecture Notes in Computer Science,* Vol. 947, Springer-Verlag, pp. 399–422.

[Steenrod 51]

STEENROD, N. *The Topology of Fibre Bundles.* Princeton University Press, 1951.

[Turski and Maibaum 87]

TURSKI, W. M., AND MAIBAUM, T. S. E. *The Specification of Computer Programs.* Addison-Wesley, 1987.

[Wadler 92]

WADLER, P. The essence of functional programming. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages* (Albuquerque, NM, January 1992), pp. 1–14.

☆U.S. GOVERNMENT PRINTING OFFICE: 1999–610–130–81116

DISTRIBUTION LIST

addresses	number of copies
DR. RAYMOND A. LIUZZI AFRL/IFTB 525 BROOKS ROAD ROME, NY 13441-4505	10
KESTREL INSTITUTE 3260 HILLVIEW AVENUE PALO ALTO, CA 94304	5
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	2
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ATTN: NAN PFRIMMER IIT RESEARCH INSTITUTE 201 MILL ST. ROME, NY 13440	1
AFIT ACADEMIC LIBRARY AFIT/LDR, 2950 P.STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1
AFRL/MLME 2977 P STREET, STE 6 WRIGHT-PATTERSON AFB OH 45433-7739	1

AFRL/HESC-TDC 1
2698 G STREET, BLDG 190
WRIGHT-PATTERSON AFB OH 45433-7604

ATTN: SMDC IM PL 1
US ARMY SPACE & MISSILE DEF CMD
P.O. BOX 1500
HUNTSVILLE AL 35807-3801

TECHNICAL LIBRARY D0274(PL-TS) 1
SPAWARSYSCEN
53560 HULL STREET
SAN DIEGO CA 92152-5001

COMMANDER, CODE 4TL000D 1
TECHNICAL LIBRARY, NAWC-WD
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6100

CDR, US ARMY AVIATION & MISSILE CMD 2
REDSTONE SCIENTIFIC INFORMATION CTR
ATTN: AMSAM-RD-OB-R, (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5000

REPORT LIBRARY 1
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545

ATTN: D'BORAH HART 1
AVIATION BRANCH SVC 122.10
FOB10A, RM 931
800 INDEPENDENCE AVE, SW
WASHINGTON DC 20591

AFIWC/MSY 1
102 HALL BLVD, STE 315
SAN ANTONIO TX 78243-7016

ATTN: KAROLA M. YOURISON 1
SOFTWARE ENGINEERING INSTITUTE
4500 FIFTH AVENUE
PITTSBURGH PA 15213

USAF/AIR FORCE RESEARCH LABORATORY
AFRL/VSOSA(LIBRARY-BLDG 1103)
5 WRIGHT DRIVE
HANSCOM AFB MA 01731-3004 1

ATTN: EILEEN LADUKE/D460
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730 1

OUSD(P)/DTSA/DUTD
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202 1

SOFTWARE ENGR'G INST TECH LIBRARY
ATTN: MR DENNIS SMITH
CARNEGIE MELLON UNIVERSITY
PITTSBURGH PA 15213-3890 1

USC-ISI
ATTN: DR ROBERT M. BALZER
4676 ADMIRALTY WAY
MARINA DEL REY CA 90292-6695 1

KESTREL INSTITUTE
ATTN: DR CORDELL GREEN
1801 PAGE MILL ROAD
PALO ALTO CA 94304 1

ROCHESTER INSTITUTE OF TECHNOLOGY
ATTN: PROF J. A. LASKY
1 LOMB MEMORIAL DRIVE
P.O. BOX 9887
ROCHESTER NY 14613-5700 1

AFIT/ENG
ATTN:TOM HARTRUM
WPAFB OH 45433-6583 1

THE MITRE CORPORATION
ATTN: MR EDWARD H. BENSLEY
BURLINGTON RD/MAIL STOP A350
BEDFORD MA 01730 1

UNIV OF ILLINOIS, URBANA-CHAMPAIGN 1
ATTN: ANDREW CHIEN
DEPT OF COMPUTER SCIENCES
1304 W. SPRINGFIELD/240 DIGITAL LAB
URBANA IL 61801

HONEYWELL, INC. 1
ATTN: MR BERT HARRIS
FEDERAL SYSTEMS
7900 WESTPARK DRIVE
MCLEAN VA 22102

SOFTWARE ENGINEERING INSTITUTE 1
ATTN: MR WILLIAM E. HEFLEY
CARNEGIE-MELLON UNIVERSITY
SEI 2218
PITTSBURGH PA 15213-38990

UNIVERSITY OF SOUTHERN CALIFORNIA 1
ATTN: DR. YIGAL ARENS
INFORMATION SCIENCES INSTITUTE
4676 ADMIRALTY WAY/SUITE 1001
MARINA DEL REY CA 90292-6695

COLUMBIA UNIV/DEPT COMPUTER SCIENCE 1
ATTN: DR GAIL E. KAISER
450 COMPUTER SCIENCE BLDG
500 WEST 120TH STREET
NEW YORK NY 10027

SOFTWARE PRODUCTIVITY CONSORTIUM 1
ATTN: MR ROBERT LAI
2214 ROCK HILL ROAD
HERNDON VA 22070

AFIT/ENG 1
ATTN: DR GARY B. LAMONT
SCHOOL OF ENGINEERING
DEPT ELECTRICAL & COMPUTER ENGRG
WPAFB OH 45433-6583

NSA/OFC OF RESEARCH 1
ATTN: MS MARY ANNE OVERMAN
9800 SAVAGE ROAD
FT GEORGE G. MEADE MD 20755-6000

AT&T BELL LABORATORIES 1
ATTN: MR PETER G. SELFRIDGE
ROOM 3C-441
600 MOUNTAIN AVE
MURRAY HILL NJ 07974

ODYSSEY RESEARCH ASSOCIATES, INC. 1
ATTN: MS MAUREEN STILLMAN
301A HARRIS B. DATES DRIVE
ITHACA NY 14850-1313

TEXAS INSTRUMENTS INCORPORATED 1
ATTN: DR DAVID L. WELLS
P.O. BOX 655474, MS 238
DALLAS TX 75265

KESTREL DEVELOPMENT CORPORATION 1
ATTN: DR RICHARD JULLIG
3260 HILLVIEW AVENUE
PALO ALTO CA 94304

DARPA/ITO 1
ATTN: DR KIRSTIE BELLMAN
3701 N FAIRFAX DRIVE
ARLINGTON VA 22203-1714

NASA/JOHNSON SPACE CENTER 1
ATTN: CHRIS CULBERT
MAIL CODE PT4
HOUSTON TX 77058

SAIC 1
ATTN: LANCE MILLER
144 WESTFIELD
MCLEAN VA 22102

STERLING IMD INC. 1
KSC OPERATIONS
ATTN: MARK MAGINN
BEECHES TECHNICAL CAMPUS/RT 26 N.
ROME NY 13440

HUGHES SPACE & COMMUNICATIONS 1
ATTN: GERRY BARKSDALE
P. O. BOX 92919
BLDG R11 MS M352
LOS ANGELES, CA 90009-2919

SCHLUMBERGER LABORATORY FOR 1
COMPUTER SCIENCE
ATTN: DR. GUILLERMO ARANGO
8311 NORTH FM620
AUSTIN, TX 78720

DECISION SYSTEMS DEPARTMENT 1
ATTN: PROF WALT SCACCHI
SCHOOL OF BUSINESS
UNIVERSITY OF SOUTHERN CALIFORNIA
LOS ANGELES, CA 90089-1421

SOUTHWEST RESEARCH INSTITUTE 1
ATTN: BRUCE REYNOLDS
6220 CULEBRA ROAD
SAN ANTONIO, TX 78228-0510

NATIONAL INSTITUTE OF STANDARDS 1
AND TECHNOLOGY
ATTN: CHRIS DABROWSKI
ROOM A266, BLDG 225
GAITHSBURG MD 20899

EXPERT SYSTEMS LABORATORY 1
ATTN: STEVEN H. SCHWARTZ
NYNEX SCIENCE & TECHNOLOGY
500 WESTCHESTER AVENUE
WHITE PLAINS NY 20604

NAVAL TRAINING SYSTEMS CENTER 1
ATTN: ROBERT BREAUX/CODE 252
12350 RESEARCH PARKWAY
ORLANDO FL 32826-3224

DR JOHN SALASIN 1
OARPA/ITO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

DR BARRY BOEHM 1
DIR, USC CENTER FOR SW ENGINEERING
COMPUTER SCIENCE DEPT
UNIV OF SOUTHERN CALIFORNIA
LOS ANGELES CA 90089-0781

DR STEVE CROSS 1
CARNEGIE MELLON UNIVERSITY
SCHOOL OF COMPUTER SCIENCE
PITTSBURGH PA 15213-3891

DR MARK MAYBURY 1
MITRE CORPORATION
ADVANCED INFO SYS TECH; G041
BURLINTON ROAD, M/S K-329
BEDFORD MA 01730

ISX 1
ATTN: MR. SCOTT FOUSE
4353 PARK TERRACE DRIVE
WESTLAKE VILLAGE,CA 91361

MR GARY EDWARDS 1
ISX
433 PARK TERRACE DRIVE
WESTLAKE VILLAGE CA 91361

DR ED WALKER 1
BBN SYSTEMS & TECH CORPORATION
10 MOULTON STREET
CAMBRIDGE MA 02238

LEE ERMAN 1
CIMFLEX TEKNOWLEDGE
1810 EMBACADERO ROAD
P.O. BOX 10119
PALO ALTO CA 94303

DR. DAVE GUNNING 1
DARPA/ISO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

DAN WELD 1
UNIVERSITY OF WASHINGTON
DEPART OF COMPUTER SCIENCE & ENGIN
BOX 352350
SEATTLE, WA 98195-2350

STEPHEN SODERLAND 1
UNIVERSITY OF WASHINGTON
DEPT OF COMPUTER SCIENCE & ENGIN
BOX 352350
SEATTLE, WA 98195-2350

DR. MICHAEL PITTARELLI 1
COMPUTER SCIENCE DEPART
SUNY INST OF TECH AT UTICA/ROME
P.O. BOX 3050
UTICA, NY 13504-3050

CAPRARO TECHNOLOGIES, INC 1
ATTN: GERARD CAPRARO
311 TURNER ST.
UTICA, NY 13501

USC/ISI 1
ATTN: BOB MCGREGOR
4676 ADMIRALTY WAY
MARINA DEL REY, CA 90292

SRI INTERNATIONAL 1
ATTN: ENRIQUE RUSPINI
333 RAVENSWOOD AVE
MENLO PARK, CA 94025

DARTMOUTH COLLEGE 1
ATTN: DANIELA RUS
DEPT OF COMPUTER SCIENCE
11 ROPE FERRY ROAD
HANOVER, NH 03755-3510

UNIVERSITY OF FLORIDA 1
ATTN: ERIC HANSON
CISE DEPT 456 CSE
GAINESVILLE, FL 32611-6120

CARNEGIE MELLON UNIVERSITY 1
ATTN: TOM MITCHELL
COMPUTER SCIENCE DEPARTMENT
PITTSBURGH, PA 15213-3890

CARNEGIE MELLON UNIVERSITY 1
ATTN: MARK CRAVEN
COMPUTER SCIENCE DEPARTMENT
PITTSBURGH, PA 15213-3890

UNIVERSITY OF ROCHESTER 1
ATTN: JAMES ALLEN
DEPARTMENT OF COMPUTER SCIENCE
ROCHESTER, NY 14627

TEXTWISE, LLC 1
ATTN: LIZ LIDDY
2-121 CENTER FOR SCIENCE & TECH
SYRACUSE, NY 13244

WRIGHT STATE UNIVERSITY 1
ATTN: DR. BRUCE BERRA
DEPART OF COMPUTER SCIENCE & ENGIN
DAYTON, OHIO 45435-0001

UNIVERSITY OF FLORIDA 1
ATTN: SHARMA CHAKRAVARTHY
COMPUTER & INFOR SCIENCE DEPART
GAINESVILLE, FL 32622-6125

KESTREL INSTITUTE 1
ATTN: DAVID ESPINOSA
3260 HILLVIEW AVENUE
PALO ALTO, CA 94304

STOLLER-HENKE ASSOCIATES 1
ATTN: T.J. GOAN
2016 BELLE MONTI AVENUE
BELMONT, CA 94002

USC/INFORMATION SCIENCE INSTITUTE 1
ATTN: DR. CARL KESSELMAN
11474 ADMIRALTY WAY, SUITE 1001
MARINA DEL REY, CA 90292

MASSACHUSETTS INSTITUTE OF TECH 1
ATTN: DR. MICHAELE SIEGEL
SLOAN SCHOOL
77 MASSACHUSETTS AVENUE
CAMBRIDGE, MA 02139

USC/INFORMATION SCIENCE INSTITUTE 1
ATTN: DR. WILLIAM SWARTHOUT
11474 ADMIRALTY WAY, SUITE 1001
MARINA DEL REY, CA 90292

STANFORD UNIVERSITY 1
ATTN: DR. GIO WIEDERHOLD
857 SIERRA STREET
STANFORD
SANTA CLARA COUNTY, CA 94305-4125

NCCOSC RDTE DIV D44208 1
ATTN: LEAH WONG
53245 PATTERSON ROAD
SAN DIEGO, CA 92152-7151

SPAWAR SYSTEM CENTER 1
ATTN: LES ANDERSON
271 CATALINA BLVD, CODE 413
SAN DIEGO CA 92151

GEORGE MASON UNIVERSITY 1
ATTN: SUSHIL JAJODIA
ISSE DEPT
FAIRFAX, VA 22030-4444

DIRNSA 1
ATTN: MICHAEL R. WARE
DOD, NSA/CSS (R23)
FT. GEORGE G. MEADE MD 20755-6000

DR. JIM RICHARDSON 1
3660 TECHNOLOGY DRIVE
MINNEAPOLIS, MN 55418

LOUISIANA STATE UNIVERSITY 1
COMPUTER SCIENCE DEPT
ATTN: DR. PETER CHEN
257 COATES HALL
BATON ROUGE, LA 70803

INSTITUTE OF TECH DEPT OF COMP SCI 1
ATTN: DR. JAIDEEP SRIVASTAVA
4-192 EE/CS
200 UNION ST SE
MINNEAPOLIS, MN 55455

GTE/BBN 1
ATTN: MAURICE M. MCNEIL
9655 GRANITE RIDGE DRIVE
SUITE 245
SAN DIEGO, CA 92123

UNIVERSITY OF FLORIDA 1
ATTN: DR. SHARMA CHAKRAVARTHY
E470 CSE BUILDING
GAINESVILLE, FL 32611-6125

AFRL/IFT 1
525 BROOKS ROAD
ROME, NY 13441-4505

AFRL/IFTM 1
525 BROOKS ROAD
ROME, NY 13441-4505

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.